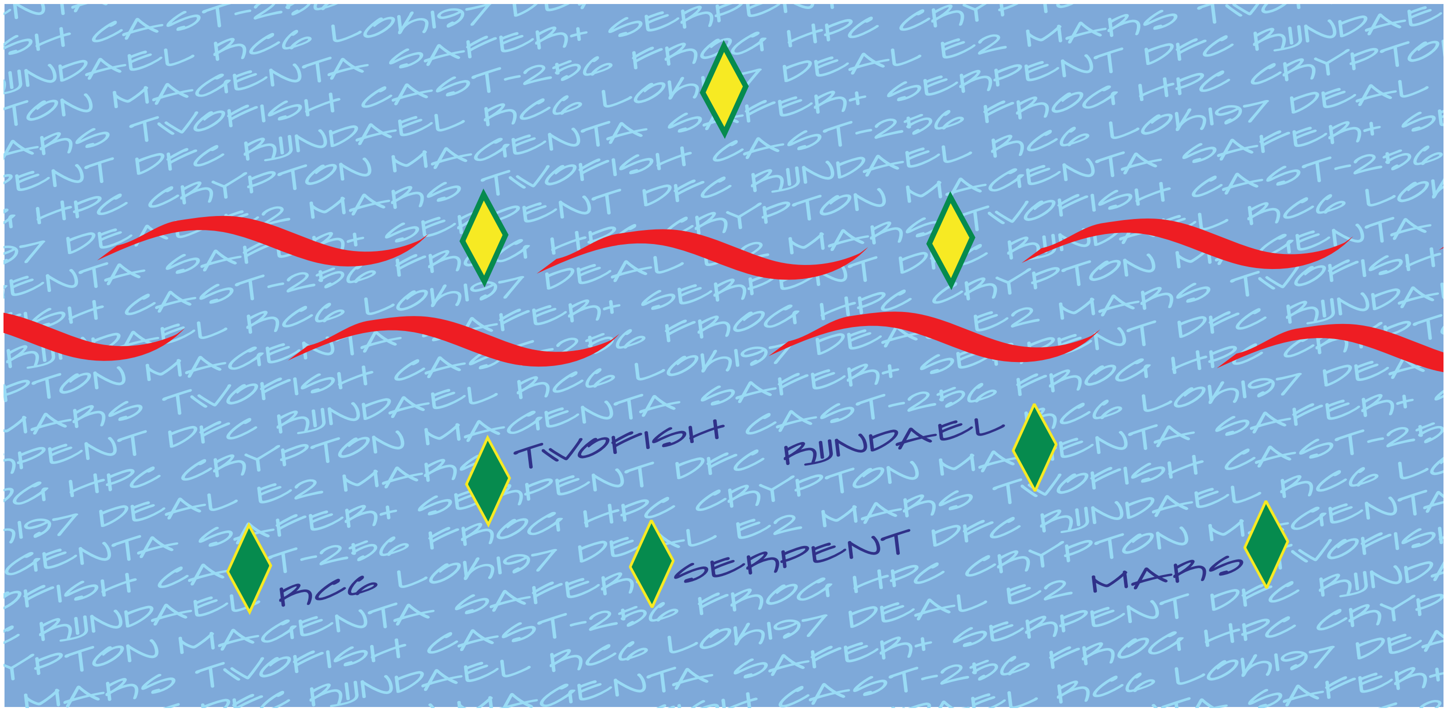
Computer Security Division

Information Technology Laboratory

AES

A Crypto Algorithm for the Twenty-first Century . . .



The Third Advanced Encryption Standard Candidate Conference

APRIL 13-14, 2000

HILTON NEW YORK AND TOWERS
New York, NY, USA

http://www.nist.gov/aes

NIST

National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

Preface

The Third Advanced Encryption Standard Candidate Conference (AES3) is the last in a series of three conferences that NIST has organized in its quest to develop the AES. It has been a long road, since NIST first announced its intention in January 1997 to develop a replacement standard for DES. Now, AES3 presents a wonderful opportunity for the cryptographic community to gather and discuss Round 2 analysis and other issues that are critical to the AES development effort. After Round 2 ends on May 15, 2000, NIST will begin the process of selecting the algorithm(s) that will be included in a draft AES Federal Information Processing Standard (FIPS). Therefore, NIST is holding AES3 to better understand which of the finalist algorithms - MARS, RC6™, Rijndael, Serpent, and Twofish - should be selected for the FIPS.

The papers to be presented at AES3 cover a wide range of issues, including cryptanalysis, implementability in Field Programmable Gate Arrays (FPGAs), hardware simulations, performance on various platforms, the role of future resiliency, and the possibility of including single or multiple algorithms in the AES FIPS.

Please see the AES home page at http://www.nist.gov/aes for the remaining papers that were proposed for AES3. Those papers - like the ones presented at AES3 - are considered official Round 2 public comments.

All Round 2 official public comments are due by May 15, 2000, and they should be submitted to AESRound2@nist.gov. This also includes any comments that interested parties may have on the papers presented at both AES3 and FSE 2000 (e.g., comments on their validity, and their applicability to and impact on the AES selection). NIST is eager to hear responses to these results and research.

The Program Committee members deserve a lot of credit for their hard work in evaluating papers, preparing for the conference, and chairing the panel presentations: Miles Smid (CygnaCom Solutions), Morris Dworkin (NIST), Tom Berson (Anagram Laboratories), Dennis Branstad (consultant, TIS Labs), Craig Clapp (PictureTel), Susan Langford (Certicom Corp.), Stefan Lucks (Universität Mannheim), Tim Moses (Entrust Technologies), and David Solo (Citigroup).

Special thanks go to the NIST staff who have provided invaluable assistance in evaluating documents and planning for AES3: Elaine Barker, Larry Bassham, Bill Burr, Jim Dray, Morris Dworkin, Jim Nechvatal, Ed Roback, and Juan Soto. Much gratitude is extended to the NIST staff responsible for the logistical side of AES3: Kathy Kilmer, Lori Phillips, and Vickie Harris.

A special mention of thanks must be made for the cooperation and assistance provided by Bruce Schneier, chair of the FSE 2000 Program Committee, and Beth Friedman of Counterpane Labs, for their efforts to coordinate these two conferences.

Finally - and most importantly - NIST greatly appreciates the efforts of all the authors who submitted papers for AES3. We have said this before, and we will say it again: the ultimate success of the AES Development Effort depends heavily on the public evaluation and analysis performed by the cryptographic community. Thank you for your hard work.

Personally, I would like to thank Miles Smid for his tireless leadership role in the AES development effort over the years, laying the solid foundation needed to support any future success that may be enjoyed by the AES.

We hope that you benefit a great deal from having joined us in New York City.

Jim Foti
NIST

April 2000

Third Advanced Encryption Standard Candidate Conference: AES3

Table of Contents

Day 1 - Thursday, April 13, 2000

Session 1: "FPGA Evaluations"

Session 2: "Platform-Specific Evaluations"

Session 3: "Surveys"

Session 4: "Cryptographic Analysis and Properties" (I)

*****

Day 2 - Friday, April 14, 2000

Session 5: "Cryptographic Analysis and Properties" (II)

Session 6: "AES Issues" Panel

Session 7: "ASIC Evaluations / Individual Algorithm Testing"

Abstracts of AES-related Papers
from the
Fast Software Encryption Workshop (FSE) 2000

Bruce Schneier
Chair, FSE 2000 Program Committee

The Seventh Fast Software Encryption Workshop (FSE 2000) was held during the three days immediately before this AES conference. Seven papers related to the AES finalists were presented at FSE 2000, and the titles and abstracts for those papers are listed below.

The proceedings for FSE 2000 will be published by Springer-Verlag in their Lecture Notes in Computer Science series. Copies of the pre-proceedings are available from the FSE secretariat.

***

Title: *Improved Cryptanalysis of Rijndael*
Authors: Niels Ferguson, John Kelsey, Bruce Schneier, Mike Stay, David Wagner, and Doug Whiting
Abstract: We improve the best attack on 6-round Rijndael from complexity 2^{72} to 2^{42}. We also present the first known attacks on 7- and 8-round Rijndael. Finally, we discuss the key schedule of Rijndael and describe a related-key technique that can break 9-round Rijndael with 256-bit keys.

Title: *On the Pseudorandomness of AES Finalists -- RC6, Serpent, MARS and Twofish*
Authors: Tetsu Iwata and Kaoru Kurosawa
Abstract: The aim of this paper is to compare the security of AES finalists in an idealized model like Luby and Rackoff. We mainly prove that a five round idealized RC6 and a three round idealized Serpent are super-pseudorandom permutations. We then show a comparison about this kind of pseudorandomness for four AES finalists, RC6, Serpent, MARS and Twofish.

Title: *Correlations in RC6*
Authors: Lars Knudsen and Willi Meier
Abstract: In this paper the block cipher RC6 is analysed. RC6 is submitted as a candidate for the Advanced Encryption Standard, and is one of five finalists. It has 128-bit blocks and supports keys of 128, 192 and 256 bits, and is an iterated 20-round block cipher. Here it is shown that versions of RC6 with 128-bit blocks can be distinguished from a random permutation with up to 15 rounds; for some weak keys up to 17 rounds. Moreover, with an increased effort key-recovery attacks can be mounted on RC6 with up to 15 rounds faster than an exhaustive search for the key.

Title: *Securing the AES Finalists Against Power Analysis Attacks*
Author: Thomas Messerges
Abstract: Techniques to protect software implementations of the AES candidate algorithms from power analysis attacks are investigated. New countermeasures that employ random masks are developed and the performance characteristics of these countermeasures are analyzed. Implementations in a 32-bit, ARM-based smartcard are considered.

Title: *Efficient Methods for Generating MARS-like S-boxes*
Authors: L. Burnett, G. Carter, E. Dawson, and W. Millan
Abstract: One of the five AES finalists, MARS, makes use of a 9x32 s-box with very specific combinatorial, differential and linear correlation properties. The s-box used in the cipher was selected as the best from a large sample of pseudo randomly generated tables, in a process that took IBM about a week to compute. This paper provides a faster and more effective alternative generation method using heuristic techniques to produce 9x32 s-boxes with cryptographic properties that are clearly superior to those of the MARS s-box, and typically take less than two hours to produce on a single PC.

Title: *A Statistical Attack on RC6*
Authors: Henri Gilbert, Helena Handschuh, Antoine Joux, and Serge Vaudenay
Abstract: This paper details the attack on RC6 which was announced in a report published in the proceedings of the second AES candidate conference (March 1999). Based on an observation on the RC6 statistics, we show how to distinguish RC6 from a random permutation and to recover the secret extended key for a fair number of rounds.

Title: *Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent*
Authors: John Kelsey, Tadayoshi Kohno, and Bruce Schneier
Abstract: We introduce a new kind of attack based on Wagner's boomerang and inside-out attacks. We first describe the new attack in terms of the original boomerang attack, and then demonstrate its use on reduced-round variants of the MARS core and of Serpent. Our attack breaks eleven rounds of the Mars core with 2^{65} chosen plaintexts, 2^{69} memory, and 2^{229} partial decryptions. Our attack breaks eight rounds of Serpent with 2^{114} chosen plaintexts, 2^{119} memory, and 2^{179} partial decryptions.

Session 1:

"FPGA Evaluations"

An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists [*]

AJ Elbirt[1], W Yip[1], B Chetwynd[2], C Paar[1]
Electrical and Computer Engineering Department
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609, USA

[1] Email: {aelbirt, waihyip, christof}@ece.wpi.edu
[2] Email: spunge@alum.wpi.edu

Abstract

The technical analysis used in determining which of the Advanced Encryption Standard candidates will be selected as the Advanced Encryption Algorithm includes efficiency testing of both hardware and software implementations of candidate algorithms. Reprogrammable devices such as Field Programmable Gate Arrays (FPGAs) are highly attractive options for hardware implementations of encryption algorithms as they provide cryptographic algorithm agility, physical security, and potentially much higher performance than software solutions. This contribution investigates the significance of FPGA implementations of four of the Advanced Encryption Standard candidate algorithm finalists. Multiple architectural implementation options are explored for each algorithm. A strong focus is placed on high throughput implementations, which are required to support security for current and future high bandwidth applications. The implementations of each algorithm will be compared in an effort to determine the most suitable candidate for hardware implementation within commercially available FPGAs.

Keywords: cryptography, algorithm-agility, FPGA, block cipher, VHDL

1 Introduction

The National Institute of Standards and Technology (NIST) has initiated a process to develop a Federal Information Processing Standard (FIPS) for the Advanced Encryption Standard (AES), specifying an Advanced Encryption Algorithm to replace the Data Encryption Standard (DES) which expired in 1998 [1]. NIST has solicited candidate algorithms for inclusion in AES, resulting in fifteen official candidate algorithms of which five have been selected as finalists. Unlike DES, which was designed specifically for hardware implementations, one of the design criteria for AES candidate algorithms is that they can be efficiently implemented in both hardware and software. Thus, NIST has announced that both hardware and software performance measurements will be included in their efficiency testing. So far, however, virtually all performance comparisons have been restricted to software implementations on various platforms [2].

The advantages of a software implementation include ease of use, ease of upgrade, portability, and flexibility. However, a software implementation offers only limited physical security, especially with respect to key storage [3] [4]. Conversely, cryptographic algorithms (and their associated keys) that are implemented in hardware are, by nature, more physically secure as they cannot easily be read or modified by an outside

[*] This research was supported in part through NSF CAREER award #CCR-9733246.

attacker [4]. The downside of traditional (ASIC) hardware implementation are the lack of flexibility with respect to algorithm and parameter switch. A promising alternative for implementation block cipher are reconfigurable hardware devices such as Field Programmable Gate Arrays (FPGAs). FPGAs are hardware devices whose function is not fixed and which can be programmed in-system. The potential advantages of encryption algorithms implemented in FPGAs include:

Algorithm Agility This term refers to the switching of cryptographic algorithms during operation. The majority of modern security protocols, such as SSL or IPsec, allow for multiple encryption algorithms. The encryption algorithm is negotiated on a per-session basis; e.g., IPsec allows among others DES, 3DES, Blowfish, CAST, IDEA, RC4 and RC6 as algorithms, and future extensions are possible. Whereas algorithm agility is costly with traditional hardware, FPGAs can be reprogrammed on-the-fly.

Algorithm Upload It is perceivable that fielded devices are upgraded with a new encryption algorithm which did not exist (or was not standardized!) at design time. In particular, it is very attractive for numerous security products to be upgraded for use of AES once the selection process is over. Assuming there is some kind of (temporary) connection to a network such as the Internet, FPGA-equipped encryption devices can upload the new configuration code.

Algorithm Modification There are applications which require modification of a standardized algorithm, e.g., by using proprietary S-boxes or permutations. Such modifications are easily made with reconfigurable hardware. Similarly, a standardized algorithm can be swapped with a proprietary one. Also, modes of operation can be easily changed.

Architecture Efficiency In certain cases, a hardware architecture can be much more more efficient if it is designed for a specific set of parameters; e.g., constant multiplication (of integers or in Galois fields) is far more efficient than general multiplication. With FPGAs it is possible to design and optimize an architecture for a specific parameter set.

Throughput Although typically slower than an ASIC implementations, FPGA implementations have the potential of running substantially faster then software implementations.

Cost Efficiency The time and costs for developing an FPGA implementation of a given algorithm are much lower than for an ASIC implementation. (However, for high-volume applications, ASIC solutions usually become the more cost-efficient choice.)

Note that algorithm agility remains an open research issue in regards to speed, physical security, and the cost associated with current high-end FPGA devices. However, we believe that cost is not a long-term limiting factor, as will be discussed in Section 3.3. For these reasons, this paper describes a thorough comparison the AES finalist algorithms RC6, Rijndael, Serpent, and Twofish with respect to implementation on state-of-the-art FPGAs. One aspect that seems to be especially relevant is the investigation of achievable encryption rates for FPGA-based implementations. We demonstrate that FPGA solutions encrypt at rates in the Gigabit range for all four algorithms investigated, which is at least one order of magnitude faster than most reported software implementations [5].

What follows is an investigation of the AES finalists to determine the nature of their underlying components. The characterization of the algorithms' components will lead to a discussion of the hardware architectures best suited for implementation of the AES finalists. A performance metric to measure the hardware cost for the throughput achieved by each algorithm's implementations will be developed and a target FPGA will be chosen so as to yield implementations that are optimized for high-throughput operation within the commercially available device. Finally, multiple architecture options of the algorithms within the targeted FPGA will be discussed and the overall performance of the implementations will be evaluated versus typical software implementations.

2 Previous Work

As opposed to custom hardware or software implementations, little work exists in the area of block cipher implementations within existing FPGAs. DES, the most common block cipher implementation targeted to FPGAs, has been shown to operate at speeds of up to 400 Mbit/s [6]. We believe that this performance can be greatly enhanced using today's technology. These speeds are significantly faster than the best software implementations of DES [7] [8] [9], which typically have throughputs below 100 Mbit/s, although a 137 Mbit/s implementation has been reported as well [7]. This performance differential is an expected result of DES having been designed in the 1970s with hardware implementations in mind.

Other block ciphers have been implemented in FPGAs with varying degrees of success. A typical example is the IDEA block cipher which has been implemented at speeds ranging from 2.8 Mbit/s [10] to 528 Mbit/s [11]. Note that while the 528 Mbit/s throughput was achieved in a fully pipelined architecture, the implementation required four Xilinx XC4000 FPGAs.

Some FPGA implementation throughputs for the AES candidates have been shown to be far slower than their software counterparts. Hardware throughputs of about 12 Mbit/s [12] [13] have been achieved for CAST-256. However, software implementations have resulted in throughputs of 37.8 Mbit/s for CAST-256 on a 200 MHz PentiumPro PC [5], a factor of three faster than FPGA implementations. When scaled to a more current 600 MHz PentiumPro PC, it is expected that the same software implementation would outperform FPGA implementations by an even larger factor. While an FPGA implementation of RC6 achieved data rates of 37.8 Mbit/s [13], our findings indicate that considerably higher data rates are achievable.

When examining the AES finalists, it is important to note that they do not necessarily exhibit similar behavior to DES when comparing hardware and software implementations. One reason for this is that the AES finalists have been designed with efficient software implementations in mind. Additionally, software implementations may be executed on processors operating at frequencies as high as 800 MHz while typical implementations that target FPGAs reach a maximum clock frequency of 50 MHz.

3 Methodology

3.1 Design Methodology

There are two basic hardware design methodologies currently available: language based (high level) design and schematic based (low level) design. Language based design relies upon synthesis tools to implement the desired hardware. While synthesis tools continue to improve, they rarely achieve the most optimized implementation in terms of both area and speed when compared to a schematic implementation. As a result, synthesized designs tend to be (slightly) larger and slower than their schematic based counterparts. Additionally, implementation results can greatly vary depending on the synthesis tool as well as the design being synthesized, leading to potentially increased variances in the synthesized results when comparing synthesis tool outputs. This situation is not entirely different from a software implementation of an algorithm in a high-level language such as C, which is also dependent on coding style and compiler quality. As shown in [14], schematic based design methodologies are no longer feasible for supporting the increase in architectural complexity evidenced by modern FPGAs. As a result, a language based design methodology was chosen as the implementation form for the AES finalists with VHDL being the specific language chosen.

3.2 Implementations — General Considerations

Each AES finalist was implemented in VHDL using a bottom-up design and test methodology. The same hardware interface was used for each of the implementations. In an effort to achieve the maximum efficiency possible, note that key scheduling and decryption were not implemented for each of the AES finalists. Because FPGAs may be reconfigured in-system, the FPGA may be configured for key scheduling and then later reconfigured for either encryption or decryption. This option is a major advantage of FPGAs implementations over classical ASIC implementations. Round keys for encryption are loaded from the external key bus and are stored in internal registers and all keys must be loaded before encryption may begin. Key loading is disabled until encryption is completed. Each implementation was simulated for functional correctness using the test vectors provided in the AES submission package [15] [16] [17] [18]. After verifying the functionality of the implementations, the VHDL code was synthesized, placed and routed, and re-simulated with annotated timing using the same test vectors, verifying that the implementations were successful.

3.3 Selection of a Target FPGA

When examining the AES finalists for hardware implementation within an FPGA, a number of key aspects emerge. First, it is obvious that the implementation will require a large amount of I/O pins to fully support the 128-bit data stream at high speeds where bus multiplexing is not an option. It is desirable to decouple the 128-bit input and output data streams to allow for a fully pipelined architecture. Since the round keys cannot change during the encryption process, they may be loaded via a separate key input bus prior to the start of encryption. Additionally, to implement a fully pipelined architecture requires 128-bit wide pipeline stages, resulting in the need for a register-rich architecture to achieve a fast, synchronous implementation. Moreover, it is desirable to have as many register bits as possible per each of the FPGA's configurable units to allow for a regular layout of design elements as well as to minimize the routing required between configurable units. Finally, it is critical that fast carry-chaining be provided between the FPGA's configurable units to maximize the performance of AES finalists that utilize arithmetic operations [13] [12].

In addition to architectural requirements, scalability and cost must be considered. We believe that the chosen FPGA should be the best chip available, capable of providing the largest amount of hardware resources as well as being highly flexible so as to yield optimal performance. Unfortunately, the cost associated with current high-end FPGAs is relatively high (several hundred US dollars per device). However, it is important to note that the FPGA market has historically evolved at an extremely rapid pace, with larger and faster devices being released to industry at a constant rate. This evolution has resulted in FPGA cost-curves that decrease sharply over relatively short periods of time. Hence, selecting a high-end device provides the closest model for the typical FPGA that will be available over the expected lifespan of AES.

Based on the aforementioned considerations, the Xilinx Virtex XCV1000BG560-4 FPGA was chosen as the target device. The XCV1000 has 128K bits of embedded RAM divided among thirty-two RAM blocks that are separate from the main body of the FPGA. The 560-pin ball grid array package provides 512 usable I/O pins. The XCV1000 is comprised of a 64 × 96 array of look-up-table based Configurable Logic Blocks (CLBs), each of which acts as a 4-bit element comprised of two 2-bit slices for a total of 12288 CLB slices [19]. This type of configuration results in a highly flexible architecture that will accommodate the round functions' use of wide operand functions. Note that the XCV1000 also appears to be a good representative for a modern FPGA and that devices from other vendors are not fundamentally different. It is thus hoped that our results carry over, within limits, to other devices.

3.4 Design Tools

FPGA Express by Synopsys, Inc. and Synplify by Synplicity, Inc. were used to synthesize the VHDL implementations of the AES finalists. As this study places a strong focus on high throughput implementations, the synthesis tools were set to optimize for speed. As will be discussed in Section 6, the resultant implementations exhibit the best possible throughputs with the associated cost being an increase in the area required in the FPGA for each of the implementations. Similarly, if the synthesis tools were set to optimize for area, the resultant implementations would exhibit reduced area requirements at the cost of decreased throughput.

XACTstep 2.1i by Xilinx, Inc. was used to place and route the synthesized implementations. For the sub-pipelined architectures, a 40 MHz timing constraint was used in both the synthesis and place-and-route processes as it resulted in significantly higher system clock frequencies. However, the 40 MHz timing constraint was found to have little affect on the other architecture types, resulting in nearly identical system clock frequencies to those achieved without the timing constraint.

Finally, Speedwave by Viewlogic Systems, Inc. and Active-HDL^{TM} by ALDEC, Inc. were used to perform behavioral and timing simulations for the implementations of the AES finalists. The simulations verified both the functionality and the ability to operate at the designated clock frequencies for the implementations.

4 Architecture Options and the AES Finalists

Before attempting to implement the AES finalists in hardware, it is important to understand the nature of each algorithm as well as the hardware architectures most suited for their implementation. What follows is an investigation into the key components of the AES finalists. Based on this breakdown, a discussion is presented on the hardware architectures most suited for implementation of the AES finalists.

4.1 Core Operations of the AES Finalist Algorithms

Algorithm	**XOR**	**Mod 2^{32} Add**	**Mod 2^{32} Subtract**	**Fixed Shift**	**Variable Rotate**	**Mod 2^{32} Multiply**	**GF(2^8) Multiply**	**LUT**
MARS	•	•	•	•	•	•		•
RC6	•	•		•	•	•		
Rijndael	•			•			•	•
Serpent	•			•				•
Twofish	•	•		•			•	•

Table 1: AES finalists core operations [20]

Modern FPGAs have a structure comprised of a two-dimensional array of configurable function units interconnected via horizontal and vertical routing channels. Configurable function units are typically comprised of look-up-tables and flip-flops. Look-up-tables may be configured as either combinational logic or memory elements. Additionally, many modern FPGAs provide variable-size SRAM blocks that may be used as either memory elements or look-up-tables [21].

In terms of complexity, the operations detailed in Table 1 that require the most hardware resources as well as computation time are the modulo 2^{32} multiplication and the variable rotation operations [20]. Implementing wide multipliers in hardware is an inherently difficult task that requires significant hardware resources. Additionally, algorithms that employ large variable rotations require a moderate amount of multiplexing hardware if carefully designed (see Section 5.1 for further discussion). S-Boxes may be implemented in either combinatorial logic or embedded RAM — the advantages of each of these options are discussed in Section 4.2. Fast operations such as bit-wise XOR, modulo 2^{32} addition and subtraction, and fixed value shifting are constructed from simple hardware elements. Additionally, the Galois field multiplications required in Rijndael and Twofish can also be implemented very efficiently in hardware as they are multiplications by a constant. Galois field constant multiplication requires far less resources than general multiplications [22].

Based on our evaluation of the AES finalists, the MARS algorithm appeared to be the most resource intensive based on its use of large S-Boxes, and modulo 2^{32} multiplication. As a result, it was conjectured that the MARS algorithm would exhibit lesser performance when compared to the other AES finalists. Due to this evaluation and a lack of development resources, the MARS algorithm was omitted from this study.

4.2 Hardware Architectures

The AES finalists are all comprised of a basic looping structure (some form of either Feistel or substitution-permutation network) whereby data is iteratively passed through a round function. Based on this looping structure, the following architecture options were investigated so as to yield optimized implementations:

- Iterative Looping
- Loop Unrolling
- Partial Pipelining
- Partial Pipelining with Sub-Pipelining

Iterative looping over a cipher's round structure is an effective method for minimizing the hardware required when implementing an iterative architecture. When only one round is implemented, an n-round cipher must iterate n times to perform an encryption. This approach has a low register-to-register delay but a requires a large number of clock cycles to perform an encryption. This approach also minimizes in general the hardware required for round function implementation but can be costly with respect to the hardware required for round key and S-Box multiplexing. Iterative looping is a subset of loop unrolling in that only one round is unrolled whereas a loop unrolling architecture allows for the unrolling of multiple rounds, up to the total number of rounds required by the cipher. As opposed to an iterative looping architecture, a loop unrolling architecture where all n rounds are unrolled and implemented as a single combinatorial logic block maximizes the hardware required for round function implementation while the hardware required for round key and S-Box multiplexing is completely eliminated. However, while this approach minimizes the number of clock cycles required to perform an encryption, it maximizes the worst case register-to-register delay for the system, resulting in an extremely slow system clock.

A partially pipelined architecture offers the advantage of high throughput rates by increasing the number of blocks of data that are being simultaneously operated upon. This is achieved by replicating the round function hardware and registering the intermediate data between rounds. Moreover, in the case of a full-length pipeline (a specific form of a partial pipeline), the system will output a 128-bit block of ciphertext at each clock cycle once the latency of the pipeline has been met. However, an architecture of this form requires significantly more hardware resources as compared to a loop unrolling architecture. In a partially pipelined architecture, each round is implemented as the pipeline's atomic unit and are separated by the registers that form the actual pipeline. However, many of the AES finalists cannot be implemented using a full-length pipeline due to the large size of their associated round function and S-Boxes, both of which must be replicated n times for an n-round cipher. As such, these algorithms must be implemented as partial pipelines. Additionally, a pipelined architecture can be fully exploited only in modes of operations which do not require feedback of the encrypted data, such as Electronic Code-Book or Counter Mode [3, Section 9.9]. When operating in feedback modes such as Ciphertext Feedback Mode, the ciphertext of one block must be available before the next block can be encrypted. As a result, multiple blocks of plaintext cannot be encrypted in a pipelined fashion when operating in feedback modes. For the remainder of our discussion, feedback mode will be abbreviated as FB and non-feedback mode will be abbreviated as NFB.

Sub-pipelining a (partially) pipelined architecture is advantageous when the round function of the pipelined architecture is complex, resulting in a large delay between pipeline stages. By adding sub-pipeline stages, the atomic function of each pipeline stage is sub-divided into smaller functional blocks. This results in a decrease in the pipeline's delay between stages. However, each sub-division of the atomic function increases the number of clock cycles required to perform an encryption by a factor equal to the number of sub-divisions. At the same time, the number of blocks of data that may be operated upon in NFB mode is increased by a factor equal to the number of sub-divisions. Therefore, for this technique to be effective, the worst case delay between stages will be decreased by a factor of m where m is the number of added sub-divisions. However, if the atomic function of the partially pipelined architecture has a small stage delay, sub-dividing the stage will achieve no significant decrease in the worst case stage delay. In this case, sub-pipelining would result in no significant increase in the system's clock frequency but would increase the logic resources and clock cycles required to perform an encryption, resulting in reduced throughput.

Many FPGAs provide embedded RAM which may be used to replace the round key and S-Box multiplexing hardware. By storing the keys within the RAM blocks, the appropriate key may be addressed based on the current round. However, due to the limited number of RAM blocks, as well as their restricted bit width, this methodology is not feasible for architectures with many pipeline stages or unrolled loops. Those architectures require more RAM blocks than are typically available. Additionally, the switching time for the RAM is more than a factor of three longer than that of a standard CLB slice element, resulting in the RAM element having a lesser speed-up effect on the overall implementation. Therefore, the use of embedded RAM is not considered for this study to maintain consistency between architectural implementations.

5 Architectural Implementation Analysis

For each of the AES finalists, the four architecture options described in Section 4.2 were implemented in VHDL using a bottom-up design and test methodology. The same hardware interface was used for each of the implementations. Round keys are stored in internal registers and all keys must be loaded before encryption may begin. Key loading is disabled until encryption is completed. These implementations yielded a great deal of knowledge in regards to the FPGA suitability of each AES finalist. What follows is a discussion of the knowledge gained regarding each algorithm when implemented using the four architecture types.

5.1 Architectural Implementation Analysis — RC6

When implementing the RC6 algorithm, it was first determined that the RC6 modulo 2^{32} multiplication was the dominant element of the round function in terms of required logic resources. Each RC6 round requires two copies of the modulo 2^{32} multiplier. However, it was found that the RC6 round function does not require a general modulo 2^{32} multiplier. The RC6 multipliers implement the function $A(2A + 1)$ which may be implemented as $2A^2 + A$. Therefore, the multiplication operation was replaced with an array squarer with summed partial products, requiring fewer hardware resources and resulting in a faster implementation. The remaining components of the RC6 round function — fixed and variable shifting, bit-wise XOR, and modulo 2^{32} addition — were found to be simple in structure, resulting in these elements of the round function requiring few hardware resources. While variable shifting operations have the potential to require considerable hardware resources, the 5-bit variable shifting required by the RC6 round function required few hardware resources. Instead of implementing a 32-to-1 multiplexor for each of the thirty-two rotation output bits (controlled by the five shifting bits), a five-level multiplexing approach was used. The variable rotation is broken into five stages, each of which is controlled by one of the five shifting bits. For each rotation output bit of a given stage, a 2-to-1 multiplexor controlled by the stage's shifting bit is used. This implementation requires a total of 160 2-to-1 multiplexors as opposed to the thirty-two 32-to-1 multiplexors required for a one-stage implementation. However, using 2-to-1 multiplexors to form the five-stage barrel-shifter results in an overall implementation that is smaller and faster when compared to the one-stage barrel-shifter implementation as described in [18, Section 3.4]. Finally, it was found that the synthesis tools could not minimize the overall size of a RC6 round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire twenty rounds of the algorithm within the target FPGA.

As discussed in Section 4.2, implementing a single round of the RC6 algorithm provides the greatest area-optimized solution. Further loop unrolling provided only minor throughput increases as the decrease in the number of cycles per encrypted block was offset by the rapidly decreasing system clock frequency. 2-stage partial pipelining was found to yield the highest throughput when operating in FB mode, outperforming the single round iterative looping implementation by achieving a significantly higher system clock frequency.

When operating in NFB mode, a partially pipelined architecture with two additional sub-pipeline stages was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, with the 10-stage partial pipeline implementation displaying the best throughput and results. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly two thirds of the round function's associated delay was attributed to the modulo 2^{32} multiplier. Therefore, two additional pipeline sub-stages were implemented so as to subdivide the multiplier into smaller blocks, resulting in a total of three pipeline stages per round function. As a result, an increase by a factor of more than 2.5 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in NFB mode. Further sub-pipelining was not implemented as this would require sub-dividing the adders used to sum the partial products (a non-trivial task) to balance the delay between sub-pipeline stages.

5.2 Architectural Implementation Analysis — Rijndael

When implementing the Rijndael algorithm, it was first determined that the Rijndael S-Boxes were the dominant element of the round function in terms of required logic resources. Each Rijndael round requires sixteen copies of the S-Boxes, each of which is an 8-bit to 8-bit look-up-table, requiring significant hardware resources. However, the remaining components of the Rijndael round function — byte swapping, constant Galois field multiplication, and key addition — were found to be simple in structure, resulting in these elements of the round function requiring few hardware resources. Additionally, it was found that the synthesis tools could not minimize the overall size of a Rijndael round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire ten rounds of the algorithm within the target FPGA.

Surprisingly, a one round partially pipelined implementation with one sub-pipeline stage provided the most area-optimized solution. As compared to a one-stage implementation with no sub-pipelining, the addition of a sub-pipeline stage afforded the synthesis tool greater flexibility in its optimizations, resulting in a more area efficient implementation. While 2-stage loop unrolling was found to yield the highest throughput when operating in FB mode, the measured throughput was within 10% of the single stage implementation. Due to the probabilistic nature of the place-and-route algorithms, one can expect a variance in performance based on differences in the starting point of the process. When performing this process multiple times, known as multi-pass place-and-route, it is likely that the single round implementation would achieve a throughput similar to that of the 2-stage loop unrolled implementation.

When operating in NFB mode, partial pipelining was found to offer the advantage of extremely high throughput rates once the pipeline latency was met, with the 5-stage partial pipeline implementation displaying the best throughput results. While Rijndael cannot be implemented using a fully pipelined architecture due to the large size of the round function, significant throughput increases were seen as compared to the loop unrolling architecture.

Sub-pipelining of the partially pipelined architectures was implemented by inserting a pipeline sub-stage within the Rijndael round function. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly half of the round function's associated delay was attributed to the S-Box substitutions. Therefore, the additional pipeline sub-stage was implemented so as to separate the S-Boxes from the rest of the round function. As a result, an increase by a factor of nearly 2 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in NFB mode. Further sub-pipelining was not implemented as this would require sub-dividing the S-Boxes (a non-trivial task) to balance the delay between sub-pipeline stages.

5.3 Architectural Implementation Analysis — Serpent

When implementing the Serpent algorithm, it was first determined that since the Serpent S-Boxes are relatively small (4-bit to 4-bit), it is possible to implement them using combinational logic as opposed to memory elements. Additionally, the S-Boxes map extremely well to the Xilinx CLB slice, which is comprised of 4-bit look-up-tables, allowing one S-Box to be implemented in a total of two CLB slices, yielding a compact implementation which minimizes routing between CLB slices. Finally, the components of the Serpent round function — key masking, S-Box substitution, and linear transformation — were found to be simple in structure, resulting in the round function requiring few hardware resources.

Implementing a single round of the Serpent algorithm provides the greatest area-optimized solution. However, a significant performance improvement was achieved by performing 8-round loop unrolling, removing the need for S-Box multiplexing hardware as one copy of each possible S-Box grouping is now included within one of the eight rounds. This amount of loop unrolling achieved a significant performance increase with little increase in hardware resources due to the compact nature of the Serpent round function. As expected, unrolling thirty-two rounds of the Serpent algorithm resulted in a lesser performance when compared to the eight round implementation. Implementing the thirty-two rounds of the algorithm in combinatorial logic severely hampered the overall clock frequency of the system, overriding the performance increase caused by the removal of the multiplexing hardware required to switch between keys.

When operating in NFB mode, a full-length pipelined architecture was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, outperforming smaller partially pipelined implementations. In the fully pipelined architecture, all of the elements of a given round function are implemented as combinatorial logic. Other AES finalists cannot be implemented using a fully pipelined architecture due to the larger round functions. However, due to the small size of the Serpent S-Boxes (4-bit look-up-tables), the cost of S-Box replication is minimal in terms of the required hardware.

Finally, sub-pipelining of the partially pipelined architectures was determined to yield no throughput increase. Because the round function components are all simple in structure, there is little performance to be gained by subdividing them with registers in an attempt to reduce the delay between stages. As a result, the increase in the system's clock frequency would not outweigh the increase in the number of clock cycles required to perform an encryption, resulting in a performance degradation.

5.4 Architectural Implementation Analysis — Twofish

When implementing the Twofish algorithm, it was first determined that the synthesis tools were unable to minimize the Twofish S-Boxes to the extent of other AES finalist algorithms due to the S-Boxes being key-dependent. Therefore, the overall size of a Twofish round was too large to allow for a fully unrolled or fully pipelined implementation of the algorithm within the target FPGA. Moreover, the key-dependent S-Boxes were found to require nearly half of the delay associated with the Twofish round function.

As expected, implementing a single round of the Twofish algorithm provides the greatest area-optimized solution in terms of total CLB slices required for the implementation. Additional loop unrolling provided minor throughput increases as the decrease in the number of cycles per encrypted block was offset by the rapidly decreasing system clock frequency. However, single stage partial pipelining with one sub-pipeline stage was found to yield the best throughput and when operating in feedback mode. With a small increases in the required hardware resources, the sub-pipelined architecture was able to reach a significantly faster system clock frequency as compared to the loop unrolling and partial pipeline implementations.

When operating in NFB mode, a partially pipelined architecture was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, with the 8-stage partial pipeline implementation displaying the best throughput results. While Twofish cannot be implemented using a fully pipelined architecture due to the large size of the round function, significant throughput increases were seen as compared to the loop unrolling architecture.

Finally, sub-pipelining of the partially pipelined architectures was implemented by inserting a pipeline sub-stage within the Twofish round function. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly half of the round function's associated delay was attributed to the S-Box substitutions. Therefore, the additional pipeline sub-stage was implemented so as to separate the S-Boxes from the rest of the round function. As a result, an increase by a factor of nearly 2 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in NFB mode. Further sub-pipelining was not implemented as this would require sub-dividing the S-Boxes (a non-trivial task) to balance the delay between sub-pipeline stages.

6 Performance Evaluation

Tables 2 and 3 detail the throughput measurements for the implementations of the three architecture types for each of the AES finalists for both NFB and FB mode. The architecture types — loop unrolling (LU), full or partial pipelining (PP), and partial pipelining with sub-pipelining (SP) — are listed along with the number of stages and (if necessary) sub-pipeline stages in the associated implementation; e.g., LU-4 implies a loop unrolling architecture with four rounds, while SP-2-1 implies a partially pipelined architecture with two stages and one sub-pipeline stage per pipeline stage. As a result, the SP-2-1 architecture implements two rounds of the given cipher with a total of two stages per round. Throughput is calculated as:

$$Throughput \ := \ \text{(128 Bits * Clock Frequency)/(Cycles Per Encrypted Block)}$$

Note that the implementation of a one stage partial pipeline architecture, an iterative looping architecture, and a one round loop unrolled architecture are all equivalent and are therefore not listed separately. Also note that the computed throughput for implementations that employ any form of hardware pipelining (as discussed in Section 4) are made assuming that the pipeline latency has been met.

The number of CLBs required as well as the maximum operating frequency for each implementation was obtained from the Xilinx report files. Note that the Xilinx tools assume the absolute worst possible operating conditions — highest possible operating temperature, lowest possible supply voltage, and worst-case fabrication tolerance for the speed grade of the FPGA [23]. As a result, it is common for actual implementations to achieve slightly better performance results than those specified in the Xilinx report files.

While this study focuses on high throughput implementations, the hardware resources required to achieve this throughput is also a critical parameter. No established metric exists to measure the hardware resource costs associated with the measured throughput of an FPGA implementation. Two area measurements of FPGA utilization are readily apparent — logic gates and CLB slices. It is important to note that the logic gate count does not yield a true measure of actual FPGA utilization. Hardware resources within CLB slices may not be fully utilized by the place-and-route software so as to relieve routing congestion. This results in an increase in the number of CLB slices without a corresponding increase in logic gates. To achieve a more accurate measure of chip utilization, CLB slice count was chosen as the most reliable area measurement. Therefore, to measure the hardware resource cost associated with an implementation's resultant throughput, the Throughput Per Slice (TPS) metric is used. We defined TPS as:

$$TPS \ := \ \text{(Encryption Rate)/(\# CLB Slices Used)}$$

Therefore, the optimal implementation will display the highest throughput and have the largest TPS. Note that the TPS metric behaves inversely to the classical time-area (TA) product.

When comparing implementations using the TPS and throughput metrics, it is required that the architectures are implemented on the same FPGA. Different FPGAs within the same family yield different timing results as a function of available logic and routing resources, both of which change based on the die size of the FPGA. Additionally, it is impossible to legitimately compare FPGAs from separate families as each family of FPGAs has a unique architecture which greatly affects the measured throughput and TPS. Finally, it is critical to note that throughput (and therefore TPS) may not scale linearly based on the number of rounds implemented for the three architecture types detailed in Section 4.1. As a result, it is imperative that multiple implementations be examined for each architecture type, varying the round count to determine the optimal number of rounds per implementation.

Algorithm	Architecture	Slices	Clock Frequency (MHz)	Cycles per Block	Throughput (Mbit/s)
RC6	LU-1	2638	13.8	20	88.5
RC6	LU-2	3069	7.3	10	94.0
RC6	LU-4	4070	3.7	5	94.8
RC6	LU-5	4476	2.9	4	92.2
RC6	LU-10	6406	1.5	2	97.4
RC6	PP-2	3189	19.8	10	253.0
RC6	PP-4	4411	12.3	5	315.5
RC6	PP-5	4848	12.1	4	386.7
RC6	PP-10	7412	13.3	2	848.1
RC6	SP-1-1	2967	26.2	20	167.6
RC6	SP-2-1	3709	26.4	10	337.8
RC6	SP-4-1	5229	24.6	5	629.8
RC6	SP-5-1	5842	25.8	4	825.2
RC6	SP-10-1	8999	26.6	2	1704.6
RC6	SP-1-2	3134	39.1	20	250.0
RC6	SP-2-2	4062	38.9	10	497.4
RC6	SP-4-2	5908	31.3	5	802.3
RC6	SP-5-2	6415	33.3	4	1067.0
RC6	**SP-10-2**	**10856**	**37.5**	**2**	**2397.9**
Rijndael	LU-1	3528	25.3	11	294.2
Rijndael	LU-2	5302	14.1	6	300.1
Rijndael	LU-5	10286	5.6	3	237.4
Rijndael	PP-2	5281	23.5	5.5	545.9
Rijndael	PP-5	10533	20.0	2.2	1165.8
Rijndael	SP-1-1	3061	40.4	10.5	491.9
Rijndael	SP-2-1	4871	38.9	5.25	949.1
Rijndael	**SP-5-1**	**10992**	**31.8**	**2.1**	**1937.9**
Serpent	LU-1	5511	15.5	32	61.9
Serpent	LU-8	7964	13.9	4	444.2
Serpent	LU-32	8103	2.4	1	312.3
Serpent	PP-8	6849	30.4	4	971.8
Serpent	**PP-32**	**9004**	**38.0**	**1**	**4860.2**
Twofish	LU-1	2666	13.0	16	104.2
Twofish	LU-2	3392	7.1	8	113.6
Twofish	LU-4	4665	3.3	4	106.8
Twofish	LU-8	6990	1.7	2	108.1
Twofish	PP-2	3519	11.9	8	190.4
Twofish	PP-4	5044	11.5	4	369.3
Twofish	PP-8	7817	10.8	2	689.5
Twofish	SP-1-1	3053	29.9	16	239.2
Twofish	SP-2-1	3869	28.6	8	457.1
Twofish	SP-4-1	5870	27.3	4	872.3
Twofish	**SP-8-1**	**9345**	**24.8**	**2**	**1585.3**

Table 2: AES finalist performance evaluation — non-feedback mode

Algorithm	**Architecture**	**Slices**	**Clock Frequency (MHz)**	**Cycles per Block**	**Throughput (Mbit/s)**
RC6	LU-1	2638	13.8	20	88.5
RC6	LU-2	3069	7.3	10	94.0
RC6	LU-4	4070	3.7	5	94.8
RC6	LU-5	4476	2.9	4	92.2
RC6	LU-10	6406	1.5	2	97.4
RC6	**PP-2**	**3189**	**19.8**	**20**	**126.5**
RC6	PP-4	4411	12.3	20	78.9
RC6	PP-5	4848	12.1	20	77.3
RC6	PP-10	7412	13.3	20	84.8
RC6	SP-1-1	2967	26.2	40	83.8
RC6	SP-2-1	3709	26.4	40	84.5
RC6	SP-4-1	5229	24.6	40	78.7
RC6	SP-5-1	5842	25.8	40	82.5
RC6	SP-10-1	8999	26.6	40	85.2
RC6	SP-1-2	3134	39.1	60	83.3
RC6	SP-2-2	4062	38.9	60	82.9
RC6	SP-4-2	5908	31.3	60	66.9
RC6	SP-5-2	6415	33.3	60	71.1
RC6	SP-10-2	10856	37.5	60	79.9
Rijndael	LU-1	3528	25.3	11	294.2
Rijndael	**LU-2**	**5302**	**14.1**	**6**	**300.1**
Rijndael	LU-5	10286	5.6	3	237.4
Rijndael	PP-2	5281	23.5	11	273.0
Rijndael	PP-5	10533	20.0	11	233.2
Rijndael	SP-1-1	3061	40.4	21	246.0
Rijndael	SP-2-1	4871	38.9	21	237.3
Rijndael	SP-5-1	10992	31.8	21	193.8
Serpent	LU-1	5511	15.5	32	61.9
Serpent	**LU-8**	**7964**	**13.9**	**4**	**444.2**
Serpent	LU-32	8103	2.4	1	312.3
Serpent	PP-8	6849	30.4	32	121.5
Serpent	PP-32	9004	38.0	32	151.9
Twofish	LU-1	2666	13.0	16	104.2
Twofish	LU-2	3392	7.1	8	113.6
Twofish	LU-4	4665	3.3	4	106.8
Twofish	LU-8	6990	1.7	2	108.1
Twofish	PP-2	3519	11.9	16	95.2
Twofish	PP-4	5044	11.5	16	92.3
Twofish	PP-8	7817	10.8	16	86.2
Twofish	**SP-1-1**	**3053**	**29.9**	**32**	**119.6**
Twofish	SP-2-1	3869	28.6	32	114.3
Twofish	SP-4-1	5870	27.3	32	109.0
Twofish	SP-8-1	9345	24.8	32	99.1

Table 3: AES finalist performance evaluation — feedback mode

Alg.	Arch.	Throughput (Gbit/s)		Slices	TPS
RC6	SP-10-2	**2.40**		10856	220881
Rijndael	SP-5-1	**1.94**		10992	176297
Serpent	PP-32	**4.86**		9004	539778
Twofish	SP-8-1	**1.59**		9345	169639

Table 4: AES finalist performance evaluation — non-feedback mode speed-optimized implementations

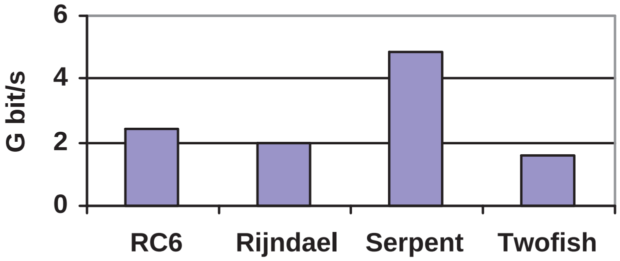


Figure 1: Best throughput — non-feedback mode

Alg.	Arch.	Throughput (Mbit/s)		Slices	TPS
RC6	PP-2	**126.5**		3189	39662
Rijndael	LU-2	**300.1**		5302	56605
Serpent	LU-8	**444.2**		7964	55771
Twofish	SP-1-1	**119.6**		3053	39169

Table 5: AES finalist performance evaluation — feedback mode speed-optimized implementations

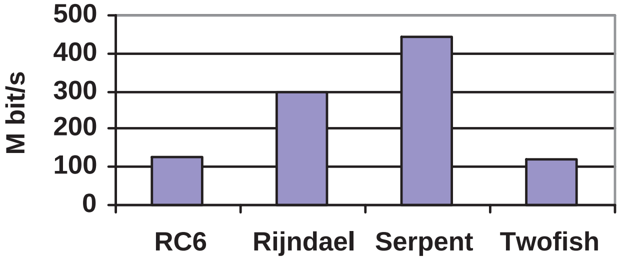


Figure 2: Best throughput — feedback mode

Tables 4 and 5 detail the optimal implementations of the AES finalists in both FB and NFB modes. Additionally, TPS is also shown for each of the implementations. It is critical to note that for the purposes of this study, the optimal implementation for an AES finalist is defined to yield the highest throughput. *As previously discussed, the synthesis tools were set to optimize for speed to guarantee that the highest throughputs would be achieved for each implementation. However, should an optimal implementation be defined based on either TPS or area, the implementation results shown in Tables 2 and 3 (and, as a result, those shown in tables 4 and 5 as well) are no longer representative of the best possible implementations for the architectures studied. To achieve a true representation that defines optimality based on either TPS or area, synthesis must be performed with the tools set to optimize for area.* While an area-efficiency analysis of the AES finalists warrants investigation, it is beyond the scope of this study.

Based on the data shown in Tables 4 and 5, the Serpent algorithm clearly outperforms the other AES finalists in both modes of operation. As compared to its nearest competitor, Serpent exhibits a throughput increase of a factor 2.2 in NFB mode and a factor 1.5 in FB mode. Interestingly, RC6, Rijndael, and Twofish all exhibit similar performance results in NFB mode. However, Rijndael exhibits significantly improved performance in FB mode as compared to RC6 and Twofish, although it is still 50% slower than Serpent.

One of the main findings of our investigation, namely that Serpent appears to be especially well suited for an FPGA implementation from a performance perspective, seems especially interesting considering that Serpent is clearly not the fastest algorithm with respect to most software comparisons [5]. Another major result of our study is that all four algorithms considered easily achieve Gigabit encryption rates with standard commercially available FPGAs. The algorithms are at least one order of magnitude faster than the best reported software realizations. These speed-ups are essentially achieved by parallelization (pipelining and sub-pipelining) of the loop structure and by wide operand processing (e.g., processing of 128 bits in once clock cycle), both of which are not feasible on current processors. We would like to stress that the pipelined architectures cannot be used to their maximum ability for modes of operation which require feedback (CFB, OFB, etc.) However we believe that for many applications which require high encryption rates, non-feedback modes (or modified feedback modes such as interleaved CFB [3, Section 9.12]) will be the modes of choice. Note that the Counter Mode grew out of the need for high speed encryption of ATM networks which required parallelization of the encryption algorithm.

7 Conclusions

The importance of the Advanced Encryption Standard and the significance of high throughput implementations of the AES finalists has been examined. A design methodology was established which in turn led to the architectural requirements for a target FPGA. The core operations of the AES finalists were identified and multiple architecture options were discussed. The implementation of each architecture option for each of the AES finalists was analyzed to determine their suitability for hardware implementation. Based on the implementation results, the best speed-optimized implementations were identified for each AES finalist in both non-feedback and feedback modes. Upon comparison, it was determined that the Serpent algorithm yielded the best performance in both modes, where best performance was defined strictly as the highest throughput. The Serpent algorithm outperforms its nearest competitor by a factor of 2.2 in non-feedback mode and by a factor of 1.5 in feedback mode.

8 Acknowledgement

We would like to thank Pawel Chodowiec and Kris Gaj from George Mason University for their helpful discussion and the VHDL code modules that were provided to assist in the implementation of some of the AES finalists. We would also like to thank Alan Martello from the University of Pittsburgh for his public-domain VHDL code module that was used in implementation of the AES finalists.

References

[1] D. Stinson, *Cryptography, Theory and Practice.* Boca Raton, FL: CRC Press, 1995.

[2] National Institute of Standards and Technology (NIST), *Second Advanced Encryption Standard (AES) Conference*, (Rome, Italy), March 1999.

[3] B. Schneier, *Applied Cryptography.* John Wiley & Sons Inc., 2nd ed., 1995.

[4] R. Doud, “Hardware Crypto Solutions Boost VPN,” *EETimes*, pp. 57–64, April 1999.

[5] B. Gladman, “Implementation Experience with AES Candidate Algorithms,” in *Proceedings: Second AES Candidate Conference (AES2)*, (Rome, Italy), March 1999.

[6] J. Kaps and C. Paar, "Fast DES Implementations for FPGAs and its Application to a Universal Key-Search Machine," in *5th Annual Workshop on Selected Areas in Cryptography (SAC '98)* (S. Tavares and H. Meijer, eds.), vol. LNCS 1556, (Queen's University, Kingston, Ontario, Canada), Springer-Verlag, August 1998.

[7] E. Biham, "A Fast New DES Implementation in Software," in *Fast Software Encryption. 4th International Workshop, FSE'97 Proceedings*, (Berlin), pp. 260–272, Springer-Verlag, 1997. Lecture Notes in Computer Science Volume 1267.

[8] A. Pfitzmann and R. Assman, "More Efficient Software Implementations of (Generalized) DES," *Computers & Security*, vol. 12, no. 5, pp. 477–500, 1993.

[9] J. Hughes, "Implementation of NBS/DES Encryption Algorithm in Software," in *Colloquium on Techniques and Implications of Digital Privacy and Authentication Systems*, 1981.

[10] D. Runje and M. Kovac, "Universal Strong Encryption FPGA Core Implementation," in *Proceedings of Design, Automation, and Test in Europe*, (Paris, France), pp. 923–924, February 1998.

[11] O. Mencer, M. Morf, and M. Flynn, "Hardware Software Tri-Design of Encryption for Mobile Communication Units," in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, (Seattle, WA), May 1998.

[12] A. Elbirt, "An FPGA Implementation and Performance Evaluation of the CAST-256 Block Cipher," Technical Report, Cryptography and Information Security Group, Electrical and Computer Engineering Department, Worcester Polytechnic Institute, Worcester, MA, May 1999.

[13] M. Riaz and H. Heys, "The FPGA Implementation of RC6 and CAST-256 Encryption Algorithms," in *accepted for CCECE'99*, (Edmonton, Alberta, Canada), 1999.

[14] C. Phillips and K. Hodor, "Breaking the 10k FPGA Barrier Calls For an ASIC-Like Design Style," *Integrated System Design*, 1996.

[15] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[16] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall, "Twofish: A 128-Bit Block Cipher," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[17] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[18] R. Rivest, M. Robshaw, R. Sidney, and Y. Yin, "The RC6$^{\mathrm{TM}}$ Block Cipher," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[19] Xilinx Inc., *Virtex 2.5V Field Programmable Gate Arrays*, 1998.

[20] B. Chetwynd, "Universal Block Cipher Module: Towards a Generalized Architectures for Block Ciphers," Master's thesis, Worcester Polytechnic Institute, Worcester, MA, November 1999.

[21] S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," in *IEEE Design & Test of Computers*, vol. 13, no. 2, pp. 42–57, 1996.

[22] C. Paar, "Optimized Arithmetic for Reed-Solomon Encoders," in *1997 IEEE International Symposium on Information Theory*, (Ulm, Germany), p. 250, June 29 – July 4 1997.

[23] P. Alfke, "Xilinx M1 Timing Parameters." electronic mail personal correspondance, December 1999.

A Comparison of the AES Candidates Amenability to FPGA Implementation

Nicholas Weaver, John Wawrzynek
{nweaver,johnw}@cs.berkeley.edu*

March 15, 2000

Abstract

The 5 final AES candidates, MARS, RC6, Rijndael, Serpent, and Twofish, are all intended to run well both on hardware and software implementations. However, the different algorithms may result in significant differences in cost and performance when implemented on FPGAs or in small custom devices. This document discusses the various algorithms from the perspective of a potential FPGA implementer. Rijndael and Twofish are excellent candidates from a hardware designer's viewpoint, while MARS is particularly expensive and inefficient.

1 Introduction

The 5 final AES candidates, MARS[2], RC6[6], Rijndael[4], Serpent[1], and Twofish[7], are all designed to run efficiently on a wide variety of hardware and software. However, the candidates vary in their amenability to hardware implementations. In this paper we estimate the relative cost and performance for various possible implementations of the different AES algorithms. Although no actual implementations are realized, it is straightforward to estimate the cost of various possible realizations of the AES candidates.

2 Observations on the candidates

The following observations will be justified throughout the paper.

MARS is not a very suitable cipher for a hardware implementation. The three separate round types, the use of both an expensive multiplier and numerous large S-box references, and a complicated subkey generation, all combine to make it a poor candidate.

RC6, though it uses comparatively expensive operations, is a reasonable candidate unless subkey generation is important. The ability to reasonably reduce the hardware requirements without sacrificing too much performance is present, a useful feature when a low cost implementation is desired. However, the subkey generation, which has a tight dependency and needs to visit elements multiple times, poses a considerable challenge for any application which needs to change keys frequency.

Rijndael is probably the best candidate when subkey flexibility isn't essential. All operations are highly parallel but comparatively inexpensive in hardware, and the subkey generation is both fast and compact. However, the additional cost of creating a separate datapath if decryption is required somewhat hampers the design, and the subkey generation may still have an impact, depending on the application.

Serpent, surprisingly enough, is not the best candidate from a hardware standpoint. Although it uses very short operations which map naturally to hardware, 32 instances of

*This work was suppored by DARPA, contract number DABT63-96-C-0048. Further support comes from the California State MICRO Program.

each of the 8 types of S-boxes quickly add up, and if a compact implementation is desired, the bandwidth is considerably reduced. Also, there is essentially no sharing between encryption and decryption pipelines.

Twofish is the best overall from a hardware viewpoint. Although not as fast as Rijndael and Serpent, the ability to perform encryption and decryption with a trivially modified pipeline is quite valuable. Also, there is a nice tradeoff space between area and performance. If subkeys are not changed, the subkey generation can largely be folded into the pipeline. If area is still tight, the pipeline can be folded in half. However, if subkeys are changed often and performance is critical, the ability to change subkeys from block to block with almost no performance penalty, whether encrypting or decrypting, is of significant potential benefit. This degree of flexibility is unique to Twofish, and is a very desirable property.

3 Possible implementation techniques

There are 3 primary criteria when measuring the candidates using a hardware metric: latency, bandwidth, and area. Latency is the amount of time required to encrypt a single block of data. If the cipher is operating in CFB or similar modes, the latency of encryption may be the critical factor. Bandwidth is the number of blocks which can be computed in a given period of time. If there is no feedback on the ciphertext, such as in ECB mode, bandwidth indicates how fast data can be encrypted. Area is a specific metric, which generally suggests the cost for an implementation. Lower area is generally beneficial, as this allows lower cost parts to be used.

The implementation fabric being considered is the Xilinx Virtex[9] Field Programmable Gate Array, which consists of an array of 4 input lookup tables (4-LUTs)[1] and associated flip flops, plus a perimeter of medium sized, dual ported, 512 byte BlockRAM memories[2]. Each 4-input lookup table can also act as a 16-bit RAM, for storing temporary values.

Embedded, small to medium sized memory blocks are becoming ubiquitous on modern FPGAs, although many older devices (such as the Xilinx 4000 series) lack such features. Thus, the use of such memories needs to be considered separately. It is, however, safe to assume that practically all future devices will have such capabilities.

It is comparatively easy to estimate the size of a hand layed out datapath for these applications, as the dataflows are suitably regular to allow the functional units to be packed together. The cost of the control logic is not considered, because for the AES candidates the primary cost is the datapath.

Similarly, the cost of generating the encryption subkeys is considered separately; for some algorithms subkey generation may be better implemented on a small microcontroller[3]. Some applications may use constant subkeys or subkeys which change only rarely, in which case subkey generation time is not a concern. However, in other applications where encryption keys may change on a packet-by-packet basis, subkey generation can become the dominant factor in the time it takes to encrypt a block.

Although the sketches described are geared towards FPGAs, a good rule of thumb is that, except for memories, logic in an FPGA takes roughly ten times the silicon area of an ASIC, while using very similar design techniques. Thus, these implementation techniques and relative cost metrics could carry over into the ASIC realm.

There are three common hardware implementation techniques considered. These are small microcoded datapaths; a pipelined, single or multiple round, C-slow[4] structure; and a fully unrolled datapath.

A microcoded datapath may be the most compact design, but often suffers from very poor bandwidth. It consists of a register file, a datapath of several functional units cus-

[1] A 4-input lookup table can realize any boolean of 4 inputs

[2] These are small, 8 address, 16b wide memories, which have two separate address and data ports. This allows two separate memory locations to be written or read in a single cycle.

[3] There is a current trend towards FPGAs with microcontrollers, such as the Triscent parts[8].

[4] Two paragraphs further defines C-slow. Be patient.

tomized to the application at hand, and a small program (usually contained in a small ROM) which controls the datapath. The problem with such implementations is that the aggregate bandwidth is usually very low and the design is unable to utilize the parallelism inherent in the algorithm.

A C-slow datapath implements a single round or group of rounds, separated into C pipeline stages which operate on different blocks. This allows for considerably higher bandwidth then a single iterative round, as C independent blocks can be processed through the pipeline. The number C is usually chosen to match the desired clock rate. A C-slow pipeline can run at a high clock rate and adding more register stages can allow an even higher clock rate (and therefore higher bandwidth) without affecting the latency[5]. Furthermore, since more operations can be done in parallel, this technique may improve the overall latency when compared with a microcoded implementation.

For most algorithms, a C-slow, single round pipeline should require roughly the same area as a microcoded datapath or an unpipelined round, while offering a considerable improvement in bandwidth, as the functional units are more highly utilized. Thus, a C-slow technique should always be utilized unless such a design simply can not be implemented in the available area or the implementation fabric is flip-flop poor.

A fully unrolled datapath, where each round is separatly implemented in hardware, can be similarly pipelined to run at a high speed. This offers essentially no latency advantage over a C-slow datapath, but allows for the maximum bandwidth possible. The number of pipeline stages is chosen in a similar way to the C-slow implementations, to provide operation which matches a target clock rate. The area cost and available bandwidth of a full pipeline are a simple multiple of the area and bandwidth for a C-slow implementation, so unrolled pipelines are not considered in detail in this analysis.

In general, we will attempt to roughly estimate the number of pipeline stages which would be required to allow a Virtex implementation to run at a 50 MHz clock cycle. A typical, modern, midsized FPGA such as the Virtex XCV200 contains 5000 4-LUTs and 14 BlockRAMs, while a typical compact, low cost FPGA such as the Xilinx Spartan2[6] XC2S50 contains 1,700 LUTs and 8 BlockRAMs.

Though these implementation sketches are for a particular FPGA fabric, these comparisons should carry over[7] to other FPGAs and small ASICs.

4 Cryptographic core

The cost for the different implementation's cryptographic cores were estimated by summarizing the costs of their respective subcomponents.

Serpent is the best from a pure performance viewpoint in hardware, although Rijndael and Twofish are close to it in performance and area/performance. The significant problem with Serpent is there is a significant minimum size for the implementation to be effective. MARS is comparatively awkward, requiring both a relatively large amount of logic and a large amount of ROM for table lookups.

The other problem with Rijndael and Serpent is that separate pipelines are required for encryption and decryption. Having to implement separate pipelines for encryption and decryption doubles the area of an implementation if both operations are required.

A RC6 pipeline can be easily modified to perform both encryption and decryption by replacing the adders with adder/subtracters (a no cost or very low-cost transformation). The Feistel basis of MARS and Twofish allow a slightly tweaked pipeline to handle both rolls effectively.

[5] This technique has a limit of the setup and hold time of the flip flop, and the granularity at which different paths may require different latencies.

[6] A low cost revision of the Virtex

[7] Although with some caveats, usually dealing with local memories and the use of tristate buffers to implement wide muxes, which may not be present in other FPGA fabrics

Algorithm	Implementation	Latency	Bandwidth	Size	Size
		(cycles)	(blocks/cycle)	(4-LUTs)	(BlockRAM)
MARS	Microcoded datapath	480	1/480	770	8
	6-Slow, single round	190	1/16	1500	12
RC6	5-Slow, single round	102	1/20	1700	0
	8-Slow, folded round	164	1/40	950	0
Rijndael	2-Slow, single round	20	1/10	780	8
Serpent	8 slow, 8 round	32	1/4	3800	0
	single round	32	1/32	1600	0
Twofish	3-Slow, single round	50	1/16	1350	0
	4-Slow, folded round	66	1/32	870	0

Figure 1: A comparison of the implementation costs for the various algorithms

A mixed Rijndael pipeline can share the S-boxes by separating the transformation from the S-box, which adds a small step and some area. However, this approach still requires a completely different column mixing step and therefore a fairly significant area cost to handle both encryption and decryption. Some implementations would probably just use separate pipelines, since depending on the implementation technology, the cost of the S-boxes may be dwarfed by the remaining costs.

Serpent can share almost no area between encryption and decryption, since it is dependent on inverse-sboxes and inverse-transformations for decryption. This essentially doubles the cost of a Serpent device which performs both encryption and decryption.

4.1 MARS

MARS is unfortunately comparatively costly to implement on small devices, as a microcoded datapath is more compact then a C-slow pipeline. There are also several comparatively expensive elements: variable rotations, the numerous, large S-box references, and the 32 bit multiplier. Since the multiplier and rotates are on the critical path and can not have their latencies hidden, a fast array multiplier and a barrel rotator are necessary to achieve good performance.

4.1.1 Microcoded datapath

A microcoded datapath would require 4 BlockRAMs for a 32 bit, 2 read, one write port register file for a scratchpad and subkey storage, another 4 BlockRAMs used as ROMs to store the S-Box, 32 LUTs for the XOR, 32 LUTs for the adder/subtracter, 160 LUTs for a barrel rotator, and 512 LUTs for an array multiplier[8]. Thus, this datapath requires roughly 8 BlockRAMs and 768 LUTs.

Assuming a single cycle latency for all operations but the multiplier, and assuming 3 cycles for the multiplier, it would take roughly 13 operations for each round of forward mixing, 18 for one round of the cryptographic core, and 12 rounds for the backwards mixing. Thus, it would require at least 480 cycles of latency for a single encryption. Furthermore, it is very difficult to run more than one or two blocks through such a datapath.

4.2 Full Round, C-slow

A C-slow pipeline is less compact on MARS when compared to other algorithms, due to the expense of various components and the 3 separate round types. The forward mixing would require 4 BlockRAMs for the S-box halves and 172 LUTs for the logic. The core would

[8] A booth encoded, shift and add multiplier could probably be constructed for only 64 to 128 LUTs, but would require 16 cycles/multiply instead of 3 cycles cycles

require roughly 4 BlockRAMs for the sbox, 64 LUTs to store the subkeys, 512 LUTs to compute R, 172 LUTs to compute M, and 172 LUTs to compute L, plus an additional 150 LUTs to compute B, C, and D, for a total of 1070 LUTs and 4 BlockRAMs for the core. The final mixing would require another 4 BlockRAMs and 172 LUTs for logic, for a total of 12 BlockRAMs and nearly 1500 LUTs. The number of 4-LUTs is reasonable, but the large number of S-box references are a considerable expense, making this implementation prohibitive on devices without local memories to use for the S-boxes.

In order to run the central core at a desired 50 MHz, it would probably be necessary to run it 6-slow[9], with the forward and backward mixings running 3 slow This would require 192 cycles of latency to encrypt a single block, but would produce one block every 16 clock cycles.

The biggest problem with MARS is the numerous references to the large S-Boxes. If a bandwidth-oriented implementation is desired, the number of S-Box references becomes very expensive. The 32 bit, modulo 2^{32} multiplier is expensive, but not prohibitively slow. Finally, the 2 variable rotations are moderately expensive operations. The biggest expense is the three different round types: although not a concern for a software implementor, it is a significant handicap for hardware designs.

4.3 RC6

RC6 uses operations which, while inexpensive in a modern microprocessor, are moderately expensive in hardware. A 32 bit, modulo 2^{32} multiplier require 512 LUTs, and a 32 bit rotator would require 160 LUTs to accomplish. However, there is a nice ability to trade off performance for area in this design.

4.3.1 Full round, C-slow

The most straightforward, compact implementation of RC6 is a single round, C-slow implementation. The initial and final keys are best stored in registers, while the remaining keys would fit in 128 LUTs. The MUXes on the start of the pipeline (to select between the input and the result from the previous round) require 128 LUTs, and the input and output whitening each require 64 LUTs.

The pipeline for the round itself would need 512 LUTs for each F function to perform the 32 bit multiplication. The variable rotations require 160 LUTs but can be combined with the XOR operation, and each of the subkey additions requires 32 LUTs. When added to the hardware required for muxing plus the initial and final adders, the total comes to roughly 1700 LUTs for the pipeline.

It should take 3 cycles to perform the F function, another cycle for the rotation, and a final cycle for the subkey addition, suggesting that a 5-slow pipeline would be sufficient. This would require 102 cycles latency to produce a result but would be able to produce a result every 20 cycles.

4.3.2 Compact, half-round, C-slow design

There are some tricks which can be used for a more compact RC6 design. Since both sides of a round are identical, the implementer could build a half-round, C-slow implementation which folds the two halves together. This roughly cuts the resource requirements and bandwidth in half, and adds three cycles of latency per round in order to exchange t and u and to perform the exchange at the end of each round, with an addititonal cycle of padding to implement a round in an even number of clock periods. In a case where bandwidth is as important as latency while resources are heavily constrained, this technique would be significantly prefered over a microcoded datapath.

[9] 3 cycles to compute R, 1 cycle to compute M, 1 to compute L, and 1 cycle to compute the new values of B, C, and D

The additional costs of such a datapath are one extra cycle for each swap and one cycle for padding, making the pipeline 8-slow and uping the latency to 164 cycles, and the bandwidth reduced to one block every 40 cycles. This allows the core to be almost cut in half, to 870 LUTs, with another 32 LUTs to store the remaining subkeys. Also, an additional 40 LUTs are required for various MUXes, and the subkey storage and whitening remain unchanged. Thus, the cost of such an implementation would be roughly 950 LUTs.

4.4 Rijndael

Rijndael's number of rounds depends on the key size. For this analysis both the block and key size are 128 bits. Rijndael has a high degree of parallelism, with very short operations and a small number of rounds, which makes it one of the fastest candidates for a hardware implementer.

The Mix-column operation of Rijndael would require 8 LUTs for the accumulation of each byte, with each multiplication probably reducable into 8 LUTs similar to the technique in [3]. Thus, the entire mix column for one 32-bit word would probably require on the order of 100 4-LUTs.

A round of Rijndael requires 8 BlockRAMs to store the S-boxes for the byte substitution[10], no area for the row shifting operation, 400 LUTs for the 4 column mixes, 128 LUTs for the key xors and the bypassing of the final column mix, 128 LUTs for the input subkey addition and pipeline MUXes, and 128 LUTs to store the subkeys.

The net result is probably 780 LUTs and 8 BlockRAMs for a single round implementation. With a critical path of 1 memory access, 3 LUTs for the column mixing, and one for the round key addition, a one or two cycle latency is reasonable for a round. With only 10 rounds of encryption, this results in an incredibly low 20 cycles of latency, with a block every 10 cycles.

Rijndael performs a greater number of rounds when used with a larger subkey. This would not affect the area required but would increase the latency and reduce the bandwidth. With 2 clock cycles for each round, it is straightforward to extrapolate the cost of a larger subkey.

4.5 Serpent

Serpent's operations, being very DES-like, map extremely well into hardware. The choice of 4 input, 4 output S-boxes allow each S-box to occupy only 4 4-LUTs, while XORs are very inexpensive, and constant rotations and permutations are free. However, although the algorithm is very fast, a considerable amount of area is required for the S-boxes which make Serpent surprisingly costly in hardware, even though its basic operations are very inexpensive.

4.5.1 Serpent 8-round, 8-slow

Due to the nature of Serpent's S-box use, the sweet spot for a serpent implementation is to unroll 8 rounds. The initial and final permutations require only wiring, not lookup tables, so the entire cost is in the encryption core.

A single round requires 128 LUTs for the key XORs, 128 LUTs to store the subkeys for the round, 128 LUTs for the S-boxes, and 160 LUTs for linear transformation, for a total of 544 LUTs for a single round. In a pipeline, a savings of 64 LUTs/round could be achieved by combining two of the key XORs with the linear transformation from the previous round, at the cost of some design complexity. For an 8 round pipeline, the total comes to 3800 LUTs for the entire pipeline.

[10] There is some wasted memory here due to the size of the BlockRAMs. Only half of the bits are actually used, which indicates that in a technology where the 8x8 S-boxes are directly implemented the area occupied would be smaller

Since each round consists only of bitwise operations and fixed rotations with a critical path of only 5 LUT evaluations, it should be pipelineable with only one cycle/round. It may even be possible to complete 1.5 to 2 rounds in a single cycle, reducing the latency further, since this critical path is so short. Thus, the 8 round pipeline would be run 8 slow, producing a result every 4 cycles, with a low latency of 32 cycles to encrypt a single block.

4.5.2 Serpent single round

A single round implementation would still need to implement all possible S-Boxes, a wide muxing step to combine the results would best be implemented with tristate buffers. Thus, 1024 LUTs would be required for the S-Boxes, 256 LUTs to store the round subkeys, 128 LUTs for the key XOR, and 160 for the linear transform. The resulting single-round implementation would require 1600 LUTs. This also introduces one more evaluation (the muxing of the S-boxes to select the correct one) into the critical path.

If pipelined at the same rate as the 8-round version, this would produce a result every 32 clock cycles, with an identical latency of 32 cycles. Since this only represents a 40% savings in area but an 8-fold reduction in bandwidth, this is not a beneficial tradeoff in most cases.

4.6 Twofish

Twofish works well in hardware without requiring memory to implement S-boxes. Though it is not the fastest or the most compact, it is reasonably small and has other advantages, including a nice area/performance tradeoff and the ability to perform encryption and decryption with a slightly modified pipeline.

The building block of Twofish, the h function, maps reasonably well to FPGA logic. Each q permutation requires 24 LUTs to implement, integrated with the S-box key XORing, for a total of 288 LUTs. The critical path is 12 LUT evaluations, short enough to expect to implement in a single cycle.

The MDS Galois matrix multiplication also maps very well. [3] shows how the multiplication by **0x5b** can be implemented in 8 LUTs, and the multiplication by **0xEF** requires 9 LUTs. It requires a further 8 LUTs to add each output together. The net result is that the matrix requires 135 LUTs to compute, with a critical path of 3 LUTs, allowing it to be combined with the PHT.

4.6.1 Twofish single round

A single round would require 846 LUTs for the two h functions, another 64 LUTs for the PHT, 64 LUTs to store the subkeys, 64 LUTs for the subkey addition, and 64 for the subkey XORing. A final 256 LUTs are required for the whitening steps, resulting in roughly 1360 LUTs for the entire pipeline.

A reasonable expectation would be for this round to take 3 cycles, one for the S-boxes, one for the MDS and PHT, and one for the key addition and XOR[11]. Such a pipeline would take 48 cycles to encrypt a single block, producing a block every 16 cycles.

4.6.2 Twofish folded

Like RC6, the symmetries in Twofish allow the pipeline to be folded in half. This would require an additional cycle to do the PHT, because the MDS would need to be split out, as well as additional logic for the PHT operation. This would require 423 LUTs for the h function, 64 LUTs for the PHT[12], 64 LUTs to store the subkeys, and 64 LUTs to perform

[11] Carries on FPGAs tend to propagate faster then the sum, but if it is necessary to develop a 3 stage pipeline, it might be best to place the pipeline in the middle of the carry of the PHT and key addition, so that the first cycle does the low 16 bits of the PHT and the key addition, and the second cycle does the high bits and the XOR operation

[12] for a 32 bit adder and the additional logic to shift or not and to select the proper input

Algorithm	Implementation	Latency (cycles)	Bandwidth (subkey sets/cycle)	Size (4-LUTs)	Size (BlockRAM)
MARS	New microcoded datapath	270	1/270	300	8
	Existing datpath modified	270	1/270	50	0
RC6	Specialized datapath	264	1/264	290	0
Rijndael	New specialized datapath	36	1/36	128	2
	Shared S-boxes	36	1/36	160	0
Serpent	8 slow, 8 round	32	1/4	2060	0
	2 slow, 2 round	32	1/16	1500	0
Twofish	Shared H-func	20	1/20	512	0
	Separate H-func	4	1/4	1260	0

Figure 2: Comparative performance and cost of subkey generation

the feistel network XOR and to rotate the output if necessary. 128 LUTs would still be needed for each of the whitening steps. It would also require an additional cycle for the PHT, in order to delay the proper element.

Such a pipeline would require roughly 870 LUTs and would require 4 cycles to complete each block, increasing the latency to 64 cycles, and reducing the bandwidth to one block every 32 cycles.

5 Subkey generation

Although subkey generation is not always on the critical path, it is may be necessary to do the subkey expansion within the device, often as a microcoded datapath or customized logic. Some applications, like point-of-sale terminals, may rarely or ever need to change their keys, in which case subkey generation isn't a priority and can be performed external to the device.

Applications such as an encrypting packet router or disk controller may require changing subkeys on a packet-by-packet or block-by-block basis. In such applications, the key setup time and parallelism may prove to be the critical factor. An important consideration for hardware implementations is how agile the key scheduling is. Being able to pipeline subkey generation at the same rate as encryption allows subkeys to be generated concurrent to encryption.

Note, though, that Rijndael and Serpent allow concurrent keyscheduling only in the encryption direction, not for decryption. These ciphers require some additional buffering for the expanded subkeys for decryption, which would make decryption latency for a changed key to be different than the encryption latency for a changed key.

The ideal case, which only occurs in Twofish, is subkeys which can be generated independently. This allows encryption and decryption subkeys to be generated on the fly regardless of whether the data is being encrypted or decrypted. This is a great advantage for devices which need to encrypt and decrypt a large number of differently keyed blocks.

In general, the datapath will only be described for a keysize of 128 bytes, if there is a significant difference in the pipeline structure for different key sizes.

Both MARS and RC6 have considerably slower subkey generation when compared with the other candidates. Neither can be effectively pipelined or accelerated, and any attempt to simultaneously produce multiple subkeys for different initial keys requires duplication of the subkey-creating hardware.

Rijndael's subkey generation is considerably shorter and takes up a small amount of area.

Although it can not be pipelined, it is small enough to duplicate if subkeys are changed often. Creating the Serpent subkeys, on the other hand, favor a heavily pipelined design due to the comparatively high cost of all the S-boxes.

Twofish's key generation can share hardware with the encryption pipeline, if a low cost implementation is required. Alternatively, it may contain it's own copy of the S-box logic and generate the subkeys concurrently with encryption, essentially eliminating all the latency involved in subkey creation.

5.1 MARS

The MARS subkey generation is best implemented in a custom microcoded datapath. If such a datapath is used for encryption, the incremental cost of subkey generation is minor, just a fair amount of expanded code with all indexes recalculated. The only addition would be a logical structure to compute M_n requiring some 100 LUTs to accomplish. If a microcoded datapath is not used, essentially the full microcoded datapath from the encryption description (sans multiplier), would be necessary, roughly 300 LUTs and 8 BlockRAMs, and roughly 270 cycles to generate the subkeys.

5.2 RC6

The RC6 subkey generation is probably best implemented with a custom datapath, using 2 BlockRAMs to store the subkeys during computation. Since the number of user key blocks is rather small, 32 LUTs used as a small RAM is sufficient. 2, 32 bit registers can store A and B, with 32 LUTs for a dedicated adder to always compute $A + B$. The only additional logic to calculate A is 2 adders, one to generate the initial value of the S array, and the second to add the current value of the S array to $A + B$, 64 LUTs in all. For updating B, this requires 160 LUTs for the rotation and 32 LUTs for another adder. Thus, the total datapath would occupy 290 LUTs.

The control logic for this structure consists only of a couple of counters and some simple state for the state machine, so it should not require significant resources.

It should be reasonable to update A in 1 cycle as it only requires 3 additions or two additions plus a memory lookup, and a constant rotation. Similarly, B should be computable in a single cycle as well. Thus, for 20 round RC6, this datapath requires 132 executions, for 264 cycles to generate the subkeys.

5.3 Rijndael

Rijndael's subkey generation is very compact. It can only produce four bytes per cycle as each word is dependent on the previous word, so an implementation which changes keys often would be still dominated by the latency of subkey generation.

Subkey generation requires 4 copies of the S-boxes in 2 BlockRAMs (either shared with the encryption pipeline or independent), enough buffering for 128b with a 128b key, 32 LUTs for the Rcon table, and 32 LUTs for the various XORs and selections. Since the buffering is dominant, the total would probably require 128 LUTs, as the flip flops end up dominating the cost. Each subkey word could be generated in a single cycle, requiring 38 cycles to generate all the subkeys.

5.4 Serpent

Just as the best Serpent implementation is an 8 round, 8 slow pipeline, the same holds for the subkey generation. Since the structure is very similar to the round itself, the same techniques can be used. It requires 64 LUTs to calculate the XORs for each of the 4 subkeys generated for each round, another 128 LUTs for the sbox substitution, and 32 LUTs for calculating the index, for 224 LUTs for each round. At 8 rounds, this comes to 1800 LUTs plus another 260 LUTs for the MUXes at the end of the pipeline, for a total of 2060 LUTs.

A more compact, 2 round design would still require 128 LUTs for the XORs, 1024 LUTs for the S-boxes with the results muxed by tristate buffers, 64 LUTs for the indexes, and 256 LUTs for the MUXes at the start of the pipeline. This would total to roughly 1500 LUTs, while still only requiring 32 cycles to generate a full set of subkeys. Although it might be possible to reuse the S-boxes from the encryption pipeline, the additional muxing would probably swamp most of the savings achieved by this reuse unless only a single-round serpent implementatino is used.

5.5 Twofish

The key generation in Twofish occurs in two parts, the first generating the two keys for the S-boxes and the second generating the round keys. The S-Box subkeys require a constant GF(2^8) matrix multiplication. Using specialization, assuming an average of 8 LUTs/constant, 256 LUTs are required to generate the subterms, and another 96 LUTs are required to perform the XORs to generate the sbox subkeys.

For implementations where subkey generation is not in the critical path one can use the S-boxes from the encryption pipeline. The modifications to the existing pipeline would add 64 LUTs to mux the inputs into the S-boxes, 64 LUTs to mux the S-box subkeys between the encryption subkeys and the input keys, and another 32 LUTs to modify the PHT, for a total of 160 LUTs, a very small addition to the pipeline. This approach would require a total of 512 LUTs of datapath to generate the subkeys and 20 cycles to generate the complete set of subkeys.

A separate round subkey datapath could be implemented, requiring an additional copy of the 2 H-functions and a PHT (910 LUTs). This would require a total area of 1260 LUTs. This is comparable to the cost of the encryption pipeline, but has the advantage that subkeys can be generated on the fly concurrently with encryption, except for those subkeys required for the input and output whitening. This allows a hardware implementation of twofish to operate at almost maximum bandwidth while able to change subkeys on a block by block basis, and to shift between encryption and decryption at will. This has the effect of reducing the key setup time to only the 4 cycles needed to generate the input and output whitening subkeys.

6 Other implementations

Twofish and Serpent have hardware implementations[3] [5] reported in the literature which can be used to help calibrate the quality of our estimates. Both implementations used HDL synthesis, which hurts performance but does not significantly affect the area required.

The Twofish implementation in [3] requires roughly 900 Xilinx 4000 CLBs, or 1800 LUTs, with the hardware for the round itself requiring roughly 1400 LUTs. A pipelined version used 7 cycles/round, running at 35 MHz. Three considerations reduced their performance: the implementation overhead of VHDL, routing congestion and tools, and an older generation FPGA.

The area numbers for this implementation are very close to the estimates for Twofish, a very good sign. Also, the performance degradation present in the HDL version is expected. HDL synthesis[13] techniques tend to produce significantly lower performing designs[14], and the Xilinx 4000 series is also significantly slower then the current generation of Xilinx FPGAs.

[13] This is where the logic is described in a High level Description Language and then compiled to form the actual circuitry of the implementation, as opposed to a lower level approach of a had specified and hand placed datapath which is assumed in the estimates.

[14] There are two factors involved: HDL synthesis on a design like Twofish is usually constructed without detailed placements for the individual modules of the datapath, and the place-and-route tools are not intelligent about placing or reconstructing datapaths in designs.

The Serpent implementation [5] is unfortunately harder to use as a calibration. It required 18,000 LUTs at 37 MHz for a fully unrolled, pipelined (1 stage/round) version, 15,000 LUTs at 13 MHz for an unpipelined, 8 round version, and 11,000 LUTs at 15 MHz for a single round when implemented in a Virtex 1000. The performance numbers are very good, and although some improvement may be achieved by a manually layed-out design, the nature of serpent doesn't have heavy datapath regularity to exploit.

The single and eight round versions can not be used to calibrate the area estimates, as the design used flipflops and MUXes for subkey storage, instead of the luts-as-memory ability present in the Virtex. Furthermore, the single round implementation used MUXes instead of the internal tristate lines to mux the S-boxes, a serious inefficiency in the implementation.

The best mechanism for attempting to calibrate area is to quadruple the area estimate for an eight round version of serpent, as a first approximation. With 15,000 LUTs for the estimated area, and 18,000 LUTs for the HDL implementation, the comparison is pretty close. The additional area for the HDL version undoubtedly includes the logic for setting the subkeys and performing I/O, while the estimate in this paper only considers the cryptographic core.

7 Conclusions and Lessons Learned

Both Rijndael and Twofish are very amenable to hardware implementations. Rijndael is the fastest, with a great degree of parallelism and very quick operations, but area requirements increase substantially if encryption and decryption is required in the same device. Although Twofish is somewhat slower, there is an excellent degree of flexibility in subkey generation and in area/performance tradeoffs.

The numerous, large S-boxes are one of the features which greatly cripple MARS hardware implementations. Having to implement 9 large 32bit S-boxes to create a single C-slow pipeline impose a significant cost on any implementation. Also, the heterogeneous round types cause a significant area penalty when compared to other implementations. The use of both S-boxes and multiplication further compounds the cost, requiring both considerable storage and considerable logic to implement.

The subkey generation for both MARS and RC6 have serial steps which require all subkeys to be modified several times. This causes subkey generation to be very slow in hardware, a significant defect when dealing with applications which require rapidly changing subkeys.

Serpent ends up being surprisingly awkward, mostly due to the large number of S-boxes required. It takes 1024 LUTs just to store all the S-boxes. Although the performance is excellent, the bandwidth quickly drops for smaller implementations and the area/performance suffers greatly.

Similarly, two operations which are cheap software, multiplication and rotation, end up being comparatively expensive in hardware. A multiplier occupies much more logic then an addition in hardware, and large multiplier are much costlier[15]. Similarly, variable rotations are much more expensive in hardware when compared to constant rotations, XORs, or additions.

Independently generated subkeys such as those in Twofish offer a great benefit for some applications, as this allows almost complete hiding of the subkey generation time. This property allows a hardware implementation to almost completely overlap subkey generation with encryption and can remove the need for any expanded subkey storage.

[15] As an example, a $32x32$ modulo 2^{32} multiplier is four times the area of a $16x16$ modulo 2^{16} multiplier.

8 Acknowledgments

Many thanks to David Wagner for explaining the design decisions and operations of various aspects of the ciphers and to Eylon Caspi for his capable editing.

References

[1] Anderson, Biham, and Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Serpent/Serpent.pdf*

[2] Burnwick *et al*, "The MARS encryption algorithm", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/MARS/mars-int.pdf*

[3] Chodowiec and Gaj, "Implementation of the Twofish Cypher Using FPGA Devices", George Mason University Techinical Report, *http://www.counterpane.com/twofish-fpga.html*

[4] Daemen and Rijmen, "AES Proposal: Rijndael", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Rijndael/Rijndael.pdf*

[5] Elbirt and Parr, "An FPGA Implementation and Performance Evaluation of the Serpent Block Ciphen", in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, February, 2000.

[6] Rivest, Robshaw, Sidney, and Yin, "The RC6 Block Cypher", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/RC6/cipher.pdf*

[7] Schneier *et al*, "Twofish: A 128-Bit Block Cypher", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Twofish/Twofish.pdf*

[8] Triscend Inc, "Triscend E5 Configurable System-on-Chip Family", *http://www.triscend.com/products/dse5csoc.pdf*

[9] Xilinx Inc, "Virtex 2.5V Field Programmable Gate Arrays", *http://www.xilinx.com/partinfo/ds003.pdf*

Comparison of the hardware performance of the AES candidates using reconfigurable hardware

Kris Gaj and Pawel Chodowiec
George Mason University
kgaj@gmu.edu, pchodowi@gmu.edu

Abstract

The results of implementations of all five AES finalists using Xilinx Field Programmable Gate Arrays are presented and analyzed. Performance of four alternative hardware architectures is discussed and compared. The AES candidates are divided into three classes depending on their hardware performance characteristics. Recommendation regarding the optimum choice of the algorithms for AES is provided.

1. Introduction

Hardware implementations of cryptography will thrive in the new century because of the growing requirements for high-speed, high-volume secure communications combined with physical security. In the presence of no major breakthroughs in cryptanalysis of the AES candidates, and relatively inconclusive results of their software performance evaluation [NBD+99, SKW+99], the comparison of the hardware performance of the AES algorithms may provide a major indicator for a final decision regarding the new standard.

Very few results regarding hardware implementations of the AES candidates have been published so far. Original documentation provided by designers of the submitted algorithms contains typically only rough estimates of the hardware performance [BCD+98, RRS+98, SKW+98]. Additionally, these estimates are very difficult to compare among each other because of large differences in assumptions regarding the technology, and because of different architecture choices. The results of actual implementations of individual algorithms, published recently by independent researchers [EP99, RH99], provide only a very fragmentary knowledge, not suitable for reliable comparison.

This situation will be certainly remedied by the publication of the NSA findings regarding hardware performance of the AES candidates. Nevertheless, the NSA evaluation plan [NSA98] targets only implementations using *semi-custom Application Specific Integrated Circuits* (ASICs), providing no data regarding other technologies. In this article, we focus on comparing AES candidates using an alternative hardware technology based on Field Programmable Gate Arrays (FPGAs). This technology, referred to as *reconfigurable hardware*, offers many advantages for future vendors and users of cryptographic equipment. It assures a short time to the market, high flexibility (including a capability for frequent modifications of hardware), low development costs, and low cost of the final product - the result of the algorithm agility - capability to use the same integrated circuit with time sharing for the execution of various secret-key and public-key algorithms. Our comparison supplements the NSA effort by covering the second primary way of implementing cryptographic algorithms in hardware.

2. Reconfigurable hardware

2.1 Operation and internal structure of an FPGA device

Field Programmable Gate Array (FPGA) is an integrated circuit that can be bought off the shelf and reconfigured by designers themselves. With each reconfiguration, which takes only a fraction of a second, an integrated circuit can perform a completely different function. FPGA consists of thousands of universal building blocks, known as *Configurable Logic Blocks* (*CLBs*), connected using programmable interconnects, as shown in Fig. 1a. Reconfiguration is able to change a function of each CLB and connections among them, leading to a functionally new digital circuit.

From several FPGA families available on the market, we have chosen for implementing AES candidates two families from Xilinx, Inc.: high performance Virtex family, and a low-cost XC4000 family. Each family consists of several FPGA devices, manufactured in the same technology, covering certain range of maximum circuit sizes.

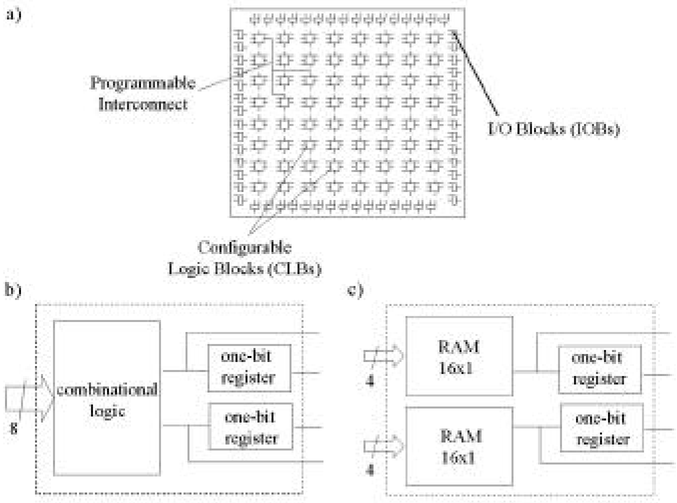


Fig. 1 FPGA device. a) General structure and main components. b) Internal structure of a CLB configured in the logic mode. c) Internal structure of a CLB configured in the memory mode.

A simplified internal structure of a CLB in the XC4000 family, and a *CLB slice* (1/2 of a CLB) in the Virtex family is shown in Figs. 1bc. In the logic mode (Fig. 1b), each of these elementary units contains a small block of combinational logic, implemented using programmable look-up tables, and two one-bit registers. In the memory mode, combinational logic is replaced by two small memories. A CLB in the XC4000 family of FPGA devices and a CLB slice in Virtex are functionally almost identical. Therefore, we will use a number of these elementary units, necessary to build a given circuit, as a measure of the circuit area and cost.

2.2 Advantages of using reconfigurable hardware for comparison of the AES candidates

For implementing cryptography in hardware, FPGAs provide the only major alternative to *custom and semi-custom Application Specific Integrated Circuits* (ASICs), integrated circuits that must be designed all the way from the behavioral description to the physical layout, and sent for an expensive and time-consuming fabrication. The comparison of the AES candidates based on FPGA devices has the following advantages over the comparison based on ASICs:

- Shorter design cycle leading to fully functioning device prototypes.
- Lower cost of the computer-aided design tools, verification, and testing.
- Potential for fast, low-cost multiple reprogramming and experimental testing of a large number of various architectures and revised versions of the same architecture.
- Higher accuracy of comparison: in the absence of the physical design and fabrication, ASIC designs are compared based on inaccurate pre-layout simulations [NSA98]; FPGA designs are compared based on very accurate post-layout simulations and experimental testing.

3. Alternative architectures

3.1 Basic organization of a block cipher implementation

The basic organization of the hardware implementation of a symmetric block cipher is shown in Fig. 2. All five AES candidates investigated in this paper can be implemented using this organization. The organization includes the following units:

a. *Encryption/decryption unit*, used to encipher and decipher input blocks of data.
b. *Key scheduling unit*, used to compute a set of internal cipher keys based on a single external key.
c. *Memory of internal keys*, used to store internal keys computed by the key scheduling unit, or loaded to the integrated circuit through the input interface.
d. *Input interface*, used to load blocks of input data and internal keys to the circuit, and to store input blocks awaiting encryption/decryption.

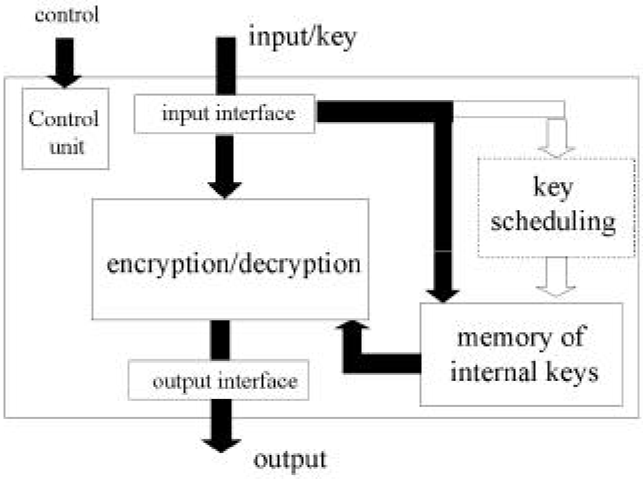


Fig. 2 Block diagram of the hardware implementation of a symmetric-block cipher.

e. *Output interface*, used to temporarily store output from the encryption/decryption unit and send it to the external memory.
f. *Control unit*, used to generate control signals for all other units.

3.2 Feedback vs. non-feedback operating modes

Today's symmetric block ciphers are used in several operating modes. From the point of view of hardware implementations, these modes can be divided into two major categories:
a. *Non-feedback modes*, such as Electronic Code Book mode (ECB), and counter mode.
b. *Feedback modes*, such as Cipher Block Chaining mode (CBC), Cipher Feedback Mode (CFB), and Output Feedback Mode (OFB).

In the non-feedback modes, encryption of each subsequent block of data can be performed independently from processing other blocks. In particular, all blocks can be encrypted in parallel. In the feedback modes, it is not possible to start encrypting the next block of data until encryption of the previous block is completed. As a result, all blocks must be encrypted sequentially, with no capability for parallel processing.

According to current security standards, the encryption of data is performed primarily using feedback modes, such as CBC and CFB. Non-feedback modes, such as ECB, are used primarily to encrypt session keys during key distribution. As a result, using current standards does not permit to fully utilize the performance advantage of the hardware implementations of secret key cryptosystems, based on parallel processing of multiple blocks of data.

3.3 Alternative architectures for the encryption/decryption unit

a. Basic architecture

The basic hardware architecture used to implement an encryption unit of a typical secret-key cipher is shown in Fig. 3a. One round of the cipher is implemented as a combinational logic, and supplemented with a single register and a multiplexer. In the first clock cycle, input block of data is fed to the circuit through the multiplexer, and stored in the register. In each subsequent clock cycle, one round of the cipher is evaluated, the result is fed back to the circuit through the multiplexer, and stored in the register. The number of clock cycles necessary to encrypt a single block of data is equal to the number of cipher rounds, #rounds.

We define the *speed* of the cipher implementation as the number of bits of data encrypted in a unit of time. Speed calculated this way is often referred to as the circuit *throughput*. The speed of the basic architecture, speed_{ba}, is given by

$$\text{speed}_{ba} = 128/\ \#\text{rounds} \cdot \text{clock\_period}\ . \qquad (1)$$

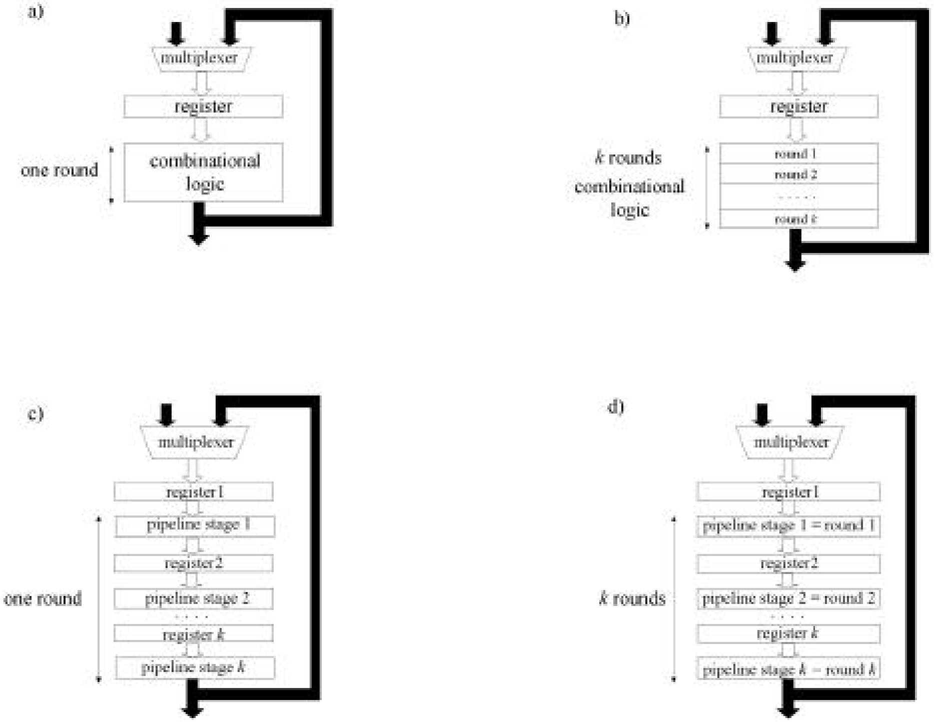


Fig. 3 Four alternative architectures for implementation of an encryption/decryption unit of a block cipher: a) basic architecture, b) architecture with the *k*-round loop unrolling, c) architecture with the *k*-stage inner-round pipelining, d) architecture with the *k*-stage outer-round pipelining.

The basic architecture combines a good speed with the relatively modest area requirements. However there exist several alternative architectures that permit to improve either one or both of these performance measures.

b. Loop unrolling

Architecture with loop unrolling is shown in Fig. 3b. The only difference compared to the basic architecture is that the combinational part of the circuit implements *k* rounds of the cipher, instead of a single round. The maximum value of *k* is equal to the number of cipher rounds. The number of clock cycles necessary to encrypt a single block of data decreases by a factor of *k*. At the same time the minimum clock period increases by a factor slightly smaller than *k*, leading to an overall relatively small increase in the cipher speed, given by

$$\text{speed}_{lu}/\text{speed}_{ba} = (1 + \tau)/(1+\tau/k), \qquad (2)$$

where τ is the ratio of the sum of the multiplexer delay, the register delay and the register setup time to the delay of a single cipher round. This increase in speed is obtained at the cost of the circuit area. Because the combinational part of the circuit constitutes the majority of the circuit area, the total area of the encryption/decryption unit increases almost proportionally to the number of unrolled rounds, *k*. Additionally, the number of internal keys used in a single clock cycle increases by a factor of *k*, which in FPGA implementations typically implies the almost proportional growth in the number of CLBs used to store internal keys.

In summary, loop unrolling enables increasing the circuit speed in both feedback and non-feedback operating modes. Nevertheless this increase is relatively small, and incurs a large area penalty.

c. Inner-round pipelining

Pipelining is a general method of increasing the amount of data processed by a digital circuit in a unit of time. The idea is to introduce evenly spaced extra registers in the middle of the combinational circuit, in such a way that several blocks of data can be processed by the circuit at the same time. Parts of the combinational logic divided by adjacent registers are called pipeline stages (see Fig. 3c). In each clock cycle the partially processed data block moves to the next pipeline stage. Its place is taken by the subsequent data block. This way, a pipelined circuit can encrypt simultaneously as many blocks of data, as the number of pipeline stages it contains.

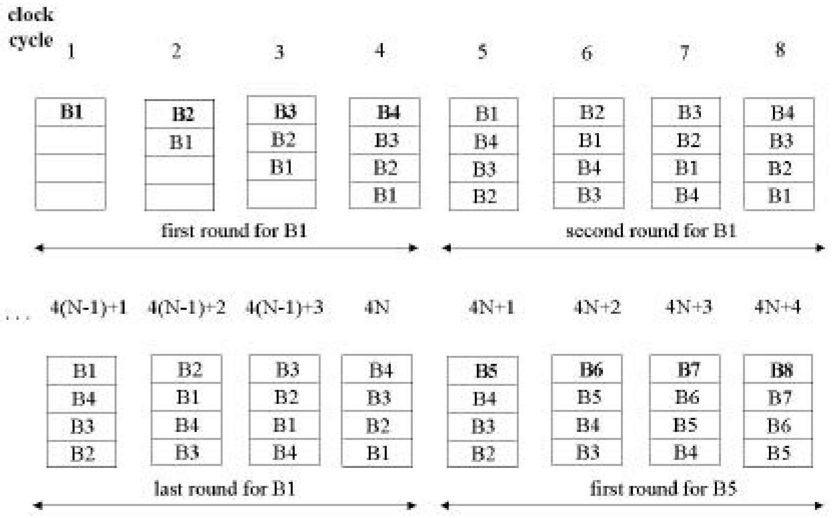


Fig. 4 Operation of the architecture with 4-stage inner-round pipelining for an N-round cipher.

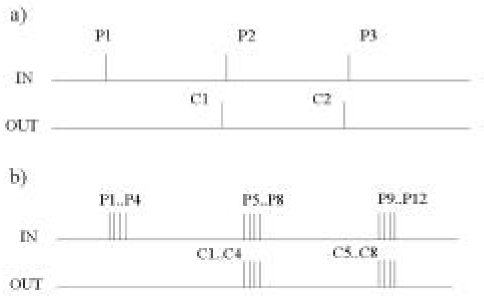


Fig. 5 Timing of input and output blocks in a) basic architecture, b) architecture with a 4-stage inner-round pipelining.

The flow of data through the pipeline during encryption is shown in Fig. 4. The number of pipeline stages in this example is four. During the first four clock cycles four subsequent blocks of data enter the pipeline. In the subsequent clock cycles, these blocks circulate in the pipeline. Each four clock cycles correspond to a single cipher round. In the cycle number 4·#rounds+1, the first block, B1, leaves the pipeline, and the fifth block, B5, is introduced to the empty pipeline stage. In the following three clock cycles, blocks B2, B3, and B4, leave the pipeline, substituted by blocks B6, B7, and B8. The timing diagram of the input and output of the circuit is shown in Fig. 5b. Speed of the circuit, expressed as the number of bits processed by the circuit in a unit of time is given by

$$\text{speed} = 128/\ \#\text{rounds} \cdot \text{reduced\_clock\_period} \qquad (3)$$

where reduced_clock_period is a minimum clock period after pipelining.

The dependence between the cipher speed-up resulting from the inner-round pipelining and the number of evenly spaced pipeline stages is shown in Fig. 6. There exists a maximum number of pipeline stages that still improves the circuit throughput. Adding additional registers will not affect the throughput. The maximum number of pipeline stages is determined by the delay of the largest indivisible combinational portion of the circuit. For majority of ciphers it is difficult to divide the cipher round into combinational stages with equal delays (especially, when the circuit is described in a high-level hardware description language, such as VHDL),

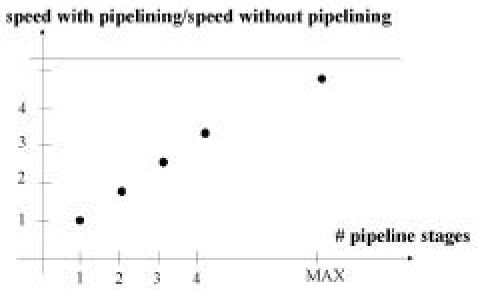


Fig. 6 Speed of the architecture with *k*-round inner-round pipelining as a function of the number of evenly spaced pipeline stages.

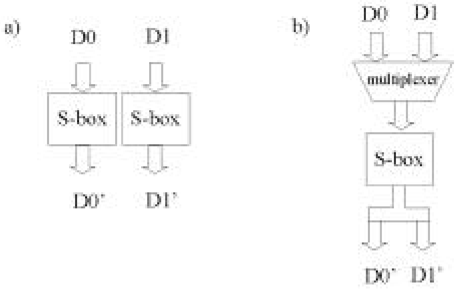


Fig. 7 Resource sharing of an S-box. a) basic operation of two parallel S-boxes, b) operation with resource sharing.

which further limits the circuit speed-up. Area of the circuit with inner-round pipelining increases only by a small percentage (area of a single 128-bit register) with each additional pipeline stage. This is especially true for FPGA circuits, in which CLBs used to implement combinational logic often contain registers not utilized in the non-pipelined implementation.

d. Outer-round pipelining

Outer-round pipelining is created by loop unrolling followed by introducing extra registers between parts of the combinational logic corresponding to each cipher round, as shown in Fig. 3d. The number of unrolled loops *k* is typically a divisor of the total number of cipher rounds, #rounds.

Area of the encryption unit with outer-round pipelining is directly proportional to the number of pipeline stages *k*. In the non-feedback cipher modes, such as ECB, the speed (throughput) of the cipher increases proportionally to the number of pipeline stages, *k*. Therefore, the outer-round pipelining enables to directly trade circuit speed with circuit area. In the feedback cipher modes, the speed of the cipher remains independent of the number of outer pipeline stages, and therefore, this kind of pipelining is not recommended for these modes.

e. Resource sharing

For some ciphers, it is possible to further decrease circuit area by time sharing of certain resources (e.g., function *h* in Twofish, 4x4 S-boxes in Serpent, 8x32 S-boxes S0, S1 in the mixing transformation of Mars, multiplication units in RC6). This is accomplished by using the same functional unit to process two (or more) parts of the data block in different clock cycles, as shown in Fig. 7b. In Fig. 7a, two parts of the data block, D0 and D1, are processed in parallel, using two independent S-boxes. In Fig. 7b, a single S-box is used to process two parts of the data block sequentially, during two subsequent clock cycles.

The use of resource sharing in real life implementations is expected to be limited, because

- Gain in the circuit area is always smaller than the loss in the circuit speed.
- The amount of area used by a basic implementation of a symmetric cipher is typically already quite small.

3.4. Choice of the figure of merit

The choice of a single figure of merit is difficult, because the optimization criteria may vary depending on the application. In our comparison, we took into account three basic figures of merit: maximum speed (throughput), minimum area, and the maximum speed/area ratio.

Optimization for maximum speed will be done in applications where communication requirements force the use of a very high speed encryption, and/or the cost of the cryptographic hardware constitutes only a small portion of the entire system. Examples of such applications include ATM and ISDN switches, Virtual Private

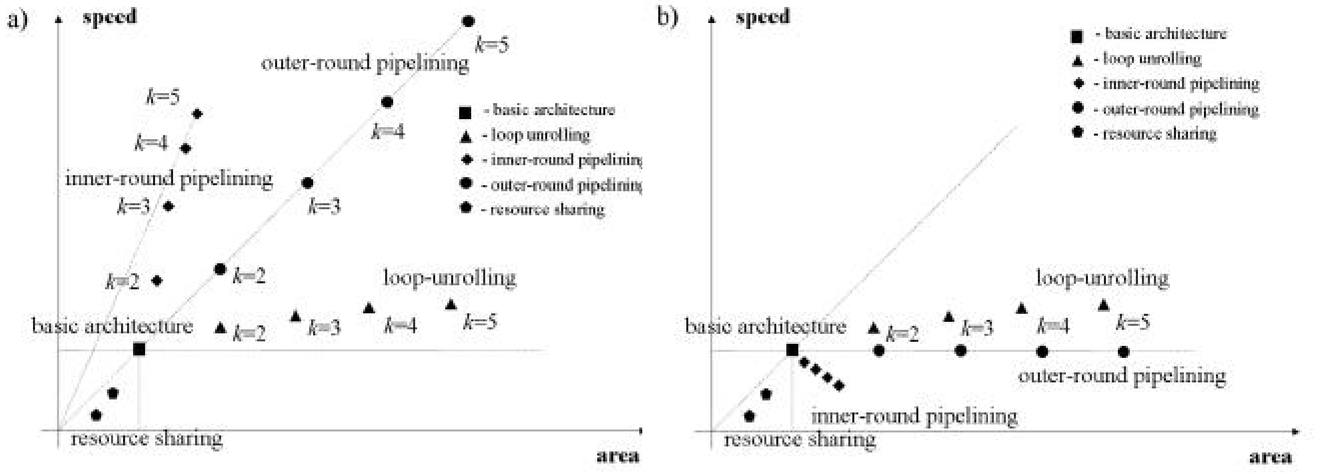


Fig. 8 Hardware performance of various alternative architectures in a) non-feedback cipher modes, such as ECB and counter mode, b) feedback cipher modes, such as CBC, CFB, and OFB.

Network routers and firewalls, WWW and database servers. In such applications, it may be justified to trade the cost of the cryptographic hardware (proportional to the circuit area) for greater speed.

In the second class of applications, the designer's goal is to obtain the maximum speed, assuming a given limit on the circuit area (cost). In such situations, the more appropriate figure of merit is the speed/area ratio. This figure of merit is particularly appropriate for non-feedback cipher modes, which enable one to directly trade circuit area for speed by using the outer-round pipelining, as shown in Fig. 8a. The examples of cost critical applications of cryptography include pagers, digital video recorders, and PCMCIA cards.

Applications that require optimization for minimum area include smart cards, embedded systems, and cellular phones. As the basic architecture may be still too big for such applications, they may enforce resource sharing. Taking into account the size and power limitations, these applications will be typically implemented using custom ASICs, not FPGAs.

3.5 Comparison of various architectures

Dependencies between the speed and the area of the encryption/decryption unit of a block cipher, for architectures discussed in section 3.3, are shown in Fig. 8.

a. Non-feedback modes

For non-feedback modes, the best speed/area ratio can be obtained by using inner-round pipelining with the maximum number of pipeline stages that still increases circuit clock frequency, as shown in Fig. 8a. The largest possible speed can be obtained by combining inner-round pipelining with outer-round pipelining. The only limit on the circuit speed is imposed in this case by the maximum circuit area (cost) and/or the maximum number of the outer-round pipeline stages (equal to the number of the cipher rounds). The smallest possible area can be obtained using the basic architecture with resource sharing.

b. Feedback-modes

For feedback modes, the basic architecture offers the best value of the ratio speed/area, as shown in Fig. 8b. Larger speed can only be obtained using loop unrolling, at the cost of a very significant increase in the circuit area (cost). Smaller area can only be obtained using resource sharing, at the cost of the significant reduction in the circuit speed.

Outer-round pipelining is inefficient in these modes, as it does not increase circuit speed, and significantly increases circuit area. Inner-round pipelining decreases speed, and increases circuit area. As a result, neither type of pipelining should be used in these operating modes.

4. Assumptions

4.1 Primary assumptions

The following tentative assumptions have been made in order to simplify the task of comparing AES candidates:

a. Key size 128 bits.

Our implementations are intended to support only one key size, 128 bits. Other key sizes required by AES (192 and 256 bits), or supported by a particular algorithm will be added in the future.

b. No key scheduling unit.

Our implementations do not support the on-chip generation of internal keys from a single external key. Instead, our implementations include a memory of internal keys loaded with the keys generated externally, and the circuitry necessary to distribute these keys from the memory to the encryption/decryption unit.

c. *Block size 128 bits.*

Only one input/output block size, 128 bits, has been considered, even if the given AES candidate supports other block sizes.

d. Basic architecture

The encryption part of all AES candidates has been implemented using basic architecture shown in Fig. 3a. This architecture has been chosen for the following reasons:

* As shown in Fig. 8b, the basic architecture assures the maximum *speed/area* ratio for feedback operating modes (CBC, CFB), now commonly used for bulk data encryption. It also guarantees near optimum speed, and near optimum area for these operating modes.

* The basic architecture is relatively easy to implement in a similar way for all AES candidates, which supports fair comparison. For architectures with inner-round pipelining, it is relatively difficult to determine and implement the maximum number of pipeline stages that still increases circuit speed and speed/area ratio.

* The implementations of the basic architecture exemplify larger differences among five AES algorithms compared to the architectures with inner-round pipelining. Inner-round pipelining permits decreasing the differences in speed among various ciphers because ciphers with longer critical path (lower speed) may be sped up by a larger factor by introducing proportionally more pipeline stages.

* Based on the performance measures for basic architecture, it is possible to derive analytically *approximate* formulas for parameters of more complex architectures, including architectures with outer-round pipelining (near proportional scaling of both area and speed), loop-unrolling (see formula (2)), and inner-round pipelining (see formula (3) and Fig. 6). Nevertheless, these formulas should be treated only as a first approximation, and the more detailed comparison requires the actual implementation of all ciphers using alternative architectures. Only such implementations may take into account the exact structure of all ciphers, limitations imposed by the FPGA architecture and the design entry method (e.g., VHDL description), and the optimization capabilities of the FPGA computer-aided design tools.

e. Resource sharing between the encryption and decryption part

In order to minimize circuit area, it was assumed that the encryption and decryption parts share as many resources as possible by the given cipher type. The effort was made to maximally decrease the effect of resource sharing on the speed of encryption and decryption.

4.2 Deviations from the basic architecture

Three ciphers, Twofish, RC6, and Rijndael, have been implemented using exactly the basic architecture shown in Fig. 3a. This was possible because all rounds of these ciphers perform exactly the same operation. For the remaining two ciphers, Serpent and Mars, this condition is not fulfilled, and as a result, small deviations from the basic architecture appeared to be necessary.

Serpent consists of 8 different rounds repeated 4 times. Therefore, it is advantageous to treat 8 official cipher rounds as a single *implementation round,* and assume that the cipher has 4 rounds. This way, 8 official cipher rounds are implemented in the basic architecture as a combinational logic. This implementation guarantees the maximum speed/area ratio typical for the basic architecture.

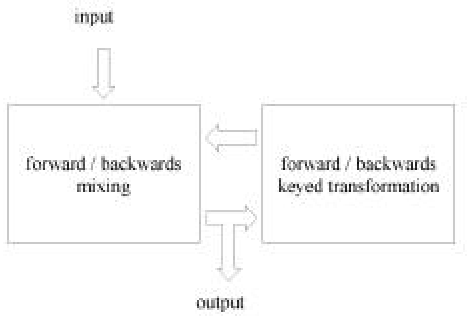

input
forward / backwards mixing
forward / backwards keyed transformation
output

Fig. 9 Deviation from the basic architecture in Mars.

In Mars, there exist four different kinds of rounds, each repeated 8 times: forward mixing, forward keyed transformation, backwards keyed transformation, and backwards mixing. It is possible to implement forward and backwards mixing using the same functional unit; the same holds for the forward and backwards keyed transformation. The structure of the mixing transformation and the keyed transformation are significantly different, and as a result they must be implemented using separate units, as shown in Fig. 9. Both of these units have an internal structure that corresponds to the basic architecture (multiplexer + register + combinational logic). Additionally, both units share the look-up table implementing two 8x32 S-boxes.

5. Results

5.1 Results for the Virtex family

The results of implementing AES candidates, according to the assumptions summarized in section 4, using the largest currently available Xilinx Virtex device, XCV1000BG560-6, are summarized in Fig. 10. For comparison, the results of implementing the current NIST standard, Triple DES, are also provided It should be stressed that all results come either from simulation or from reports generated by Xilinx tools, and have not as yet been confirmed experimentally. The details of all implementations, including the detailed block diagrams, and the description of simulation and test experiments will be provided in the technical report available at the AES conference [CG00]. Part of this report, describing Twofish, is already available on the web [CG99].

Implementations of all ciphers take from 9% (for Twofish) to 38% (for Serpent) of the total number of 12288 CLB slices available in the Virtex device used in our designs. It means that less expensive Virtex devices could be used for all implementations. Additionally, the key scheduling unit can be easily implemented within the same device as the encryption/decryption unit.

5.2 Results for the XC4000 family

For the low-cost, medium-size family of Xilinx FPGA devices, XC4000, only two ciphers, Twofish and RC6, were able to fit within the largest device from this family. The relative performance of these ciphers is similar to the relative performance in Virtex implementations. It is interesting to notice that for the two different FPGA devices from this family, the smaller one guarantees the higher speed.

Cipher	**Speed [Mbit/s]**		**Area [CLBs]**		**Speed/Area [kbit/s·CLB]**	
	4028/4036	4085	4028/4036	4085	4028/4036	4085
Twofish	90.9	89.2	907	907	100.2	98.3
RC6	45.9	43.1	1222	1222	37.6	35.3

Table I. Results of implementing Twofish and RC6 using the largest available FPGA device from the XC4000XL family, XC4085XL, and the largest device fitting the implementation of the respective cipher, i.e., XC4028XL for Twofish, and XC4036XL for RC6.

5.3 Resource sharing between encryption and decryption

The amount of resource sharing between encryption and decryption is considerably different for various AES candidates, depending on the type of the cipher. Resource sharing is close to 100% for Feistel ciphers and modified Feistel ciphers, and close to zero for S-P networks. The level of resource sharing can be described by the amount and type of the extra logic that must be added to the circuit implementing encryption, so that the modified circuit can perform both encryption and decryption, as shown in Table II.

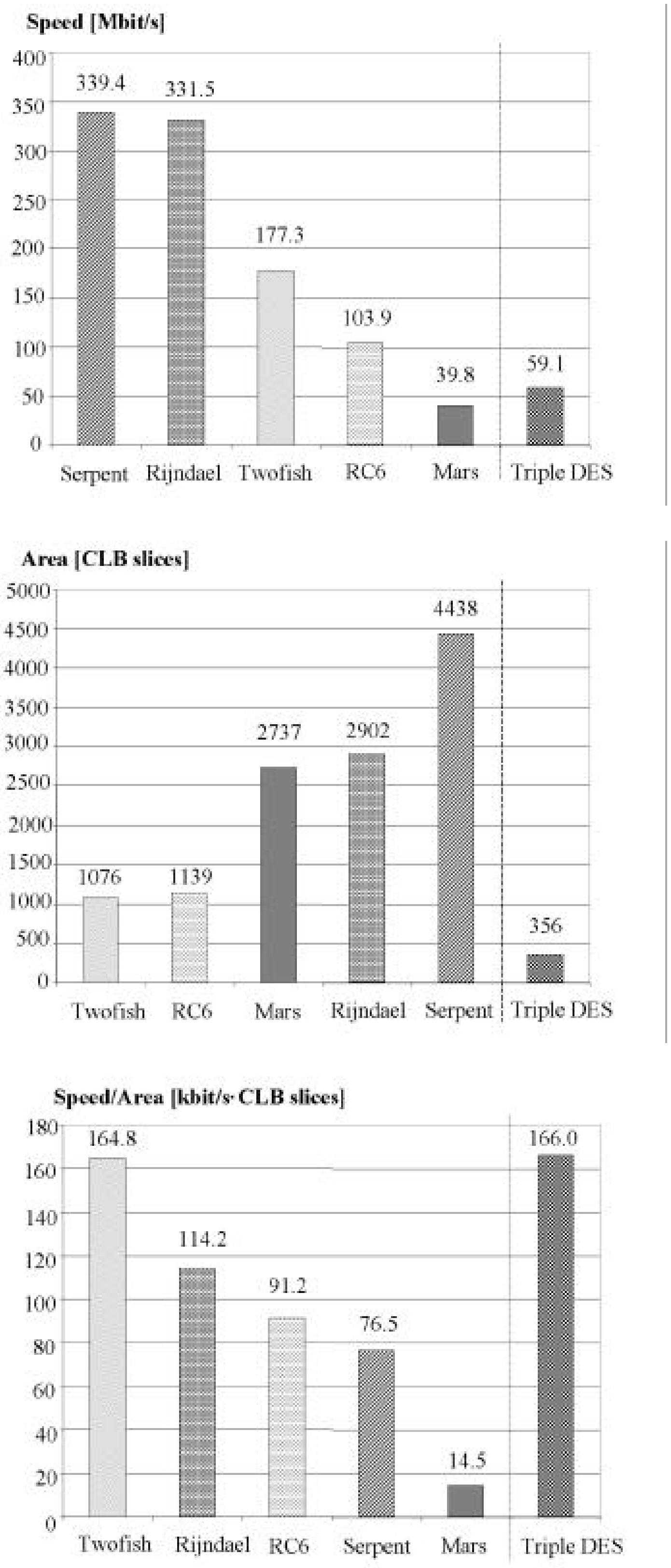


Fig. 10 Results of implementing AES candidates using Xilinx Virtex FPGA devices.

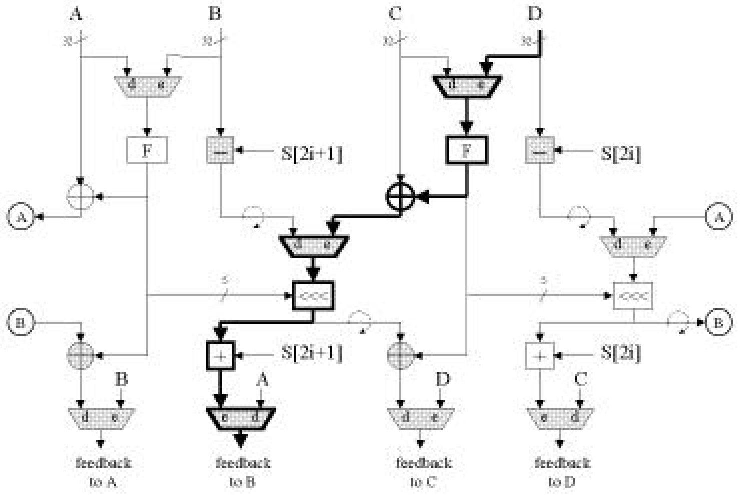


Fig. 11 Combinational part of a single round of RC6 implemented using basic architecture. Shaded components had to be added to the encryption unit, so it could perform decryption. The thick line shows the critical path in the circuit. Unit F performs operation $(2(X^2 \bmod 2^{32})+ X) \bmod 2^{32} <<< 5$. An arrow around a line means inverting the order of bits.

The relative size of the extra circuitry is the smallest for Mars and Twofish (less than 10%), and about 20% for RC6 (see Fig. 11). For Serpent and Rijndael, encryption and decryption are performed by two independent units of equal size. For Rijndael, these two units share 16 look-up tables implementing inversions in the Galois Field GF(2^8). These look-up tables take about 45% of the area used for encryption. Thus, the extra decryption circuitry takes for Serpent 100%, and for Rijndael about 55% of the area required for encryption itself.

Cipher	Extra logic	Extra logic area /encryption logic area
Twofish	2 32-bit XOR2, 2 32-bit MUX2	6%
Mars	2 SUB32, 3 32-bit MUX2	3%
RC6	2 SUB32, 2 32-bit XOR2, 8 32-bit MUX2 (see Fig. 11)	20%
Rijndael	Decryption independent of encryption, except 16 S-boxes 8x8	55%
Serpent	Decryption independent of encryption	100%

Table II. Extra logic that must be added to the circuit implementing encryption, so that the modified circuit can perform both encryption and decryption. Notation: XOR2 - 2-input XOR, MUX2 - 2-input multiplexer, SUB32 - 32-bit subtractor.

5.4 Critical path

The critical paths of all five AES candidates are characterized in Table III. As an example, the critical path of RC6 (without init MUX) is shown in Fig. 11.

Based on the characteristics of the critical path, the AES candidates can be divided into two main categories. Ciphers from the first category, RC6 and Mars, include in the critical path one complex arithmetic operation, such as modular multiplication or modular squaring, which determines the minimum clock period of these ciphers. The second category includes Rijndael, Twofish, and Serpent. In these ciphers, the critical path includes one or several S-boxes, and several multiple-input XORs. The minimum clock period is the sum of the access time to memories used to implement S-boxes, and delays introduced by multiple-input XORs and other simple auxiliary operations. The critical path of Twofish contains additionally two 32-bit additions.

The effect of resource sharing between encryption and decryption on the critical path is the strongest for RC6 (three encryption/decryption multiplexers in the critical path), very small for Rijndael, Twofish and Mars (one encryption/decryption multiplexer in the critical path), and negligible for Serpent. In Mars, additional delay (2 multiplexers) is caused by sharing resources between the forward and backwards keyed transformations.

Cipher	Minimum clock period - Virtex [ns]	Minimum clock period - XC4000 [ns]	Number of rounds	Components in the critical path (path flow / list of operations)
Rijndael	38.6	-	10	E/D MUX → S-box → affine transformation → MixColumn → init MUX
				S-box 8x8, XOR6, XOR5, XOR4, XOR2, 2 MUX2
Twofish	45.1	88.0	16	S-box → MDS → PHT → key addition → xor → E/D MUX → init MUX
				6 S-box 4x4, 2 ADD32, 9 XOR2, XOR4, XOR5, 2 MUX2
Serpent	94.3	-	4	8 x {key mixing → S-box → linear transformation) → init MUX
				8 S-box 4x4, 8 XOR2, 8 XOR7, MUX2
RC6	61.6	139.5	20	E/D MUX → squaring → addition → xor → E/D MUX → variable rotation → addition → E/D MUX → init MUX
				SQR32, 2 ADD32, ROT32, XOR2, 4 MUX2
Mars	100.6	-	32	2 mode MUXes → E/D MUX → multiplication → XOR → init MUX
				MUL32, XOR2, 4 MUX2

Table III. Critical paths in the implementation of the basic architecture for all AES candidates. Notation:
E/D MUX - encryption/decryption multiplexer, i.e., multiplexer used to change the data flow between encryption and decryption; mode MUX - multiplexer used to change the data flow depending on the mode of transformation (e.g., forward and backwards transformation in Mars); init MUX - multiplexer used to select between loading a new block of data and feeding back data from the end of the cipher round (the only multiplexer shown in Fig. 3a); XOR*n* - *n*-input XOR, MUX2 - 2-input multiplexer, ADD32 - 32-bit adder, MUL32 - 32-bit multiplier mod 2^{32}, SQR32 - 32-bit squaring mod 2^{32}, ROT32 - variable rotation of a 32-bit word.

5.5 Area critical components

The components contributing most to the circuit area, for each AES candidate, are shown in Table IV. The ciphers fall clearly into two groups: Twofish and RC6 have the area approximately three to four times smaller than the area of the remaining three candidates, Mars, Rijndael, and Serpent. The relatively small area of Twofish and RC6 comes from the fact that both ciphers are of the Feistel type. The relatively large size of Serpent and Rijndael comes from the fact that both ciphers are S-P networks, and the amount of resource sharing between encryption and decryption is limited (no resource sharing for Serpent, about 45% resource sharing for Rijndael). Additional factor contributing to the large size of Serpent is the use of eight different types of S-boxes in eight subsequent cipher rounds.

Cipher	# of CLB slices - Virtex	# of CLBs - XC4000	Area critical components
Twofish	1076	907	96 S-box 4x4 (6 kbit), 18 32-bit XOR2, 24 MUL GF(2^8)
RC6	1139	1222	2 SQR32, 12 32-bit MUX2, 2 ROT32
Serpent	4438	-	512 S-box 4x4 (32 kbit), 2048 XORn (linear transformation, n=2..7)
Mars	2737	-	4 S-box 8x32 (32 kbit), MUL32, 22 32-bit MUX2
Rijndael	2902	-	16 S-box 8x8 (32 kbit), 24 MUL GF(2^8), 256 XOR5 (affine and inverse affine transformation)

Table IV. Cipher components contributing most to the circuit area. Notation: MUL GF(2^8) - multiplication in the Galois Field GF(2^8), XOR*n* - *n*-input XOR, MUX2 - 2-input multiplexer, MUL32 - 32-bit multiplier mod 2^{32}, SQR32 - 32-bit squaring mod 2^{32}, ROT32 - variable rotation of a 32-bit word.

The relatively large size of Mars is the result of the design decisions, such as

a. using two different kinds of rounds (mixing vs. keyed transformation). For the basic non-pipelined architecture, only one type of round is active at a time.
b. using 4 large S-boxes 8x32 in a single round of the mixing transformation. Sharing two of these S-boxes during mixing transformation is possible only at the cost of doubling the number of clock cycles required for this transformation. (Our implementation still shares two S-boxes between the mixing transformation and the keyed transformation.)
c. using area-consuming 32x32 bit modular multiplication.

The area of Mars, Serpent, and Rijndael is dominated by S-boxes. Even though the number and size of these S-boxes is very different for each cipher, the total number of bits in memories implementing S-boxes, 32 kbits, is identical for all three ciphers. This may explain the relatively similar size of all three implementations expressed in number of CLBs.

5.6 Potential for inner-round pipelinig

Inner round pipelining can be most effectively applied to the ciphers with the following features:

a. the cipher round is composed of a large number of layers, with all layers performing simple operations with comparable delays;
b. the cipher round does not contain large hard-to-divide functional units.

Additionally, for FPGA implementations, it is advantageous if the implementation of the basic architecture contains large number of CLBs with unused flip-flops (one bit registers).

The above conditions are the best fulfilled by Serpent. It is straightforward to introduce 8 internal pipeline stages to the implementation round of Serpent (one implementation round = 8 regular cipher rounds), one after each regular cipher round. Implementing pipeline stages inside of the regular cipher round is possible in theory, but may be difficult in practice because of the clock frequency limitations imposed by the control unit.

The second cipher best suited for inner-round pipelining is Twofish. According to Table III, the critical path of Twofish contains a large number of simple operations with comparable delays, including a 4x4 S-box read-out, XOR operations, and additions. The most complex of these operations is a 32-bit addition. It is likely that this operation may need to be implemented using multilevel carry-lookahead architecture to take the full advantage of the inner-round pipelining in Twofish. Additionally, the FPGA implementation of basic architecture of Twofish contains a relatively small number of unused flip-flops, which will cause that the circuit area will increase by a larger percentage than for Serpent with the same number of inner-round pipeline stages.

Rijndeal is relatively easy to pipeline, but its critical path contains only 7 elementary operations. Additionally, the most time-consuming of these operations, the 8x8 S-box read-out, is hard to divide into extra pipeline stages. RC6 can be efficiently pipelined at the cost of increase in the circuit area resulting from using fast architectures for addition and multiplication (e.g., carry lookahead and carry save). Mars is the most difficult to pipeline because of the

a. irregular structure with different operations in various paths;
b. two types of rounds (mixing and keyed transformation) both using large S boxes;
c. need for the complex fast architectures for the pipelined multiplication and addition.

5.7 Potential for loop unrolling

The largest gain from loop unrolling can be achieved by ciphers with the following properties:

* small area used by the combinational part of a single round, which permits fitting a large amount of rounds in the largest available FPGA device;
* small delay of a single round compared to the sum of delays eliminated by loop unrolling, including the round multiplexer delay, the register delay, and the register setup time (as shown in formula (2)).
* potential for optimizations at the boundary between the last and the first operation of the cipher round.

Assuming the use of the largest available Virtex chip, RC6 and Twofish have the highest potential for loop unrolling. The largest Virtex chip can easily fit ten RC6 rounds and eight Twofish rounds. Mars can be implemented with four rounds unrolled; Rijndael and Serpent with only two rounds unrolled.

5.8 Potential for outer-round pipelining and mixed outer-inner-round pipelining

The largest gain from outer-round pipelining can be achieved by ciphers with the smallest area. The largest number of pipelined rounds fitting within the largest available Virtex chip is the same as in the architecture with loop unrolling. As a result, Twofish and RC6 can benefit most from the outer-round pipelined architecture. The throughput of both these ciphers exceeds 1 Gbit/s for the architectures with the maximum number of outer-round pipeline stages. Additional speed-up can be obtained by combining outer and inner round pipelining, leading to the mulitigigabit-per-second performance. For Serpent, the most straightforward form of mixed pipelining, with 16 regular cipher rounds unrolled and a register after each regular cipher round (1/8 of the implementation round), would result in an even higher performance. Mars can benefit substantially from both forms of pipelining; Rijndael primarily from the inner-round pipelining.

6. Design procedure and tools

The design flow and tools used in our group for implementation of algorithms in FPGA devices are shown in Fig. 12. All five AES ciphers were first described in VHDL, and their description verified using the functional simulator from Aldec, Inc. Test vectors and intermediate results from the reference software implementations were used for debugging and verification of VHDL codes. The revised VHDL code became an input to Xilinx tools performing the automated logic synthesis, mapping, placing, and routing. These tools generated reports describing the area and speed of implementations, a netlist used for timing simulations, and a bitstream to be used to program an actual FPGA device. A final step is to verify the design experimentally, using physical FPGA devices. We plan to perform these experiments using a PCI FPGA board from Virtual Computer Corporation [VCC]. The most complex PCI board currently available from VCC is based on the XC4062XL FPGA device. This device is able to fit full implementations of Twofish and RC6, and an encryption portion of Serpent. All details of our implementations and experiments will be described in the technical report [CG00].

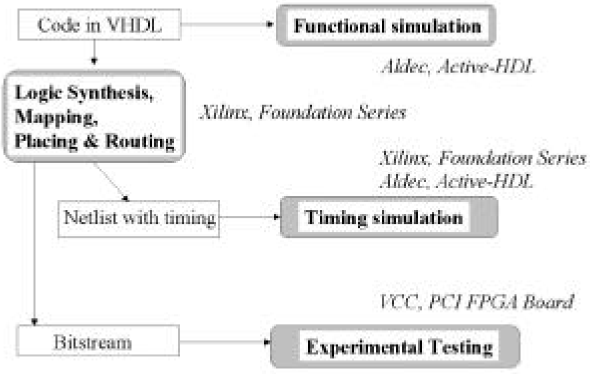


Fig. 12 Design flow for implementing AES candidates using Xilinx FPGA devices.

7. Need for interleaved operating modes

The full potential of hardware implementations of symmetric block ciphers can only be utilized in cipher modes that support efficient use of pipelining, as shown in Fig. 8. To date, the ECB mode is the only operating mode standardized by NIST that supports efficient pipelining. However, ECB is not regarded secure for transmissions of large volumes of data, and most standard protocols recommend using CBC or CFB modes instead. Therefore, we propose to speed-up the standardization effort, and include in the AES standard interleaved modes of operation, such as the interleaved CBC mode defined by:

$$C_i = \text{AES}(M_i \oplus IV_i) \text{ for } i=1 \text{ to } N, \text{ and } C_i = \text{AES}(M_i \oplus C_{i-N}) \text{ for } i>N\ . \tag{4}$$

The standard should support arbitrary values of the interleaving factor N, smaller than a certain maximum.

8. Conclusions

The results and analyses presented in this paper show that the differences in hardware performance of the AES candidates are bigger and more significant than the corresponding differences in software performance. No correlation between software and hardware performance was found. On the contrary, Serpent, believed to be the slowest candidate in software, appeared to be the fastest of the five AES candidates in hardware. We believe that the large differences among parameters of all five AES algorithms in hardware resulted primarily from internal structure of these algorithms, and were not significantly affected by our implementation decisions. On the other hand, we could not completely eliminate or predict the influence of the FPGA design tools and the VHDL design entry method on the results of the comparison. Assessed exclusively from the hardware performance point of view, the five AES finalists fall into the three distinct classes with different performance characteristics.

The first class includes Twofish and RC6. Both ciphers guarantee compact low-cost implementations with medium speed compared to other candidates. In particular, because of the area constraints, Twofish and RC6 are the only ciphers that can be implemented using low cost FPGA devices from the Xilinx XC4000 family. Both ciphers can be substantially sped-up by outer-round pipelining (for non-feedback modes (ECB, counter mode)), and - to the lesser extent - by loop-unrolling (for cipher feedback modes (CBC, CFB)). Among the two, Twofish is in some respects superior to RC6. It is about 70% faster and is more suitable for inner-round pipelining. Both ciphers use comparable area, and as a result their potential for loop unrolling and outer-round pipelining is similar.

The second class includes Serpent and Rijndael. Both ciphers guarantee very high speed at the cost of the relatively large area compared to the ciphers from the first class. The primary way of speeding up these ciphers for non-feedback cipher modes (ECB and counter mode) is inner-round pipelining. Both ciphers have a similar speed in the basic architecture. Rijndael can be implemented using about 35% less area. The more regular architecture of Serpent makes it significantly more suitable for a multi-stage inner-round pipelining.

The third class is composed of Mars itself. This cipher shows the worst hardware characteristics of all five candidates. It is over twice as slow than the next slowest candidate (RC6), and over 8 times slower than the fastest AES cipher (Serpent). It also takes over twice the area used by ciphers from the first group, Twofish and RC6. Further optimizations of the Mars implementation are certainly possible, but would require the higher development effort than that devoted to other AES candidates.

It is interesting to notice that although four out of five candidates outperform Triple DES in terms of speed, only Twofish has a comparable performance in terms of the speed/area ratio. Three other candidates, Rijndael, RC6, and Serpent, have a similar, and much lower than triple DES, value of this performance parameter.

Out of all five candidates, Twofish seems to be the most suitable for applications where the primary requirement is the limited cost or area of the cryptographic hardware. Serpent and Rijndael both offer superior performance for applications where the speed itself is a criterion of primary concern.

Acknowledgments

The authors would like to thank Christof Paar and his students, as well as Miles Smid and other members of the NIST Computer Security Division for valuable comments and discussions. We also would like to thank our students, Po Khuon and Tanvir Joy, for their work on implementation of Triple DES.

Literature:

[BCD+98] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, "Mars - A Candidate Cipher for AES," NIST AES Proposal, June 1998.

[CG99] P. Chodowiec and K. Gaj, "Implementation of the Twofish Cipher Using FPGA Devices", Technical Report, George Mason University, July 1999; available at http://www.counterpane.com/twofish.html.

[CG00] P. Chodowiec and K. Gaj, "Implementations of the AES Candidate Algorithms using FPGA Devices," Technical Report, George Mason University, April 2000 (to be published on the web).

[EP99] A.J. Elbirt and C. Paar, "An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher," Eighth ACM International Symposium on Field-Programmable Gate Arrays, Monterey, California, February 10-11, 2000. Preprint available at http://ece.wpi.edu/Research/crypt/publications/index.html.

[NBD+99] James Nechvatal, Elaine Barker, Donna Dodson, Morris Dworkin, James Foti, Edward Roback, "Status Report on the First Round of the Development of the Advanced Encryption Standard," NIST report, August 1999.

[NSA98] National Security Agency, "Initial plans for estimating the hardware performance of AES submissions," http://csrc.nist.gov/encryption/aes/round2/round2.htm.

[RH99] M. Riaz and H. Heys, "The FPGA Implementation of RC6 and CAST-256 Encryption Algorithms," accepted for CCECE'99, Edmonton, Alberta, Canada, 1999.

[RRS+98] R. Rivest, M. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 Block Cipher," NIST AES Proposal, June 1998.

[SKW+98] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, "Twofish: A 128-Bit Block Cipher," NIST AES Proposal, June 1998.

[SKW+99] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, "Performance Comparison of the AES Submissions," Second AES Candidate Conference, Rome, April 1999.

[VCC] Virtual Computer Corporation, http://www.vcc.com/

Session 2:

"Platform-Specific Evaluations"

AES Finalists on PA-RISC and IA-64: Implementations & Performance

John Worley, Bill Worley, Tom Christian, Christopher Worley[1]
Hewlett Packard Labs
Fort Collins, CO

Overview

The Advanced Encryption Standard selection process has, for the first time, included software execution speed as a relevant criterion for the choice of the next standard. The initial submissions included keying, encryption, and decryption execution times, in clock cycles, for Intel Pentium, Pentium II, and Pentium Pro microprocessors. While Pentium execution speeds are important, by no means do they completely characterize software performance, particularly that of existing RISC microprocessors and the new IA-64 microprocessor family.

In order to enable a more complete characterization of software performance, our group, working from HP Labs, decided in January 1999, to study and publish the performance of likely AES finalists for PA-RISC and IA-64 microprocessors. We initially selected RC6, Rijndael, Serpent, and Twofish. Our preliminary results were informally presented at the 1999 Rome Conference. Following the selection of the five finalists, we included work on MARS. This paper discusses the issues, implementations, and results of our work for each of the five AES finalists.

Details of specific engineering tradeoffs for Itanium and McKinley chips remain proprietary. We therefore are not at liberty to disclose complete source codes and performance details from which such information can be deduced. What we have chosen to present are actual simulation cycle counts for a snapshot of the evolving McKinley design. These are not cycle counts for an actual product. We offer them as well-substantiated, conservative indicators of the performance of the future family of IA-64 processors. Itanium will be somewhat slower; future implementations will be faster. We believe these results do provide a reasonable basis for software performance judgments about the AES finalists. A summary table appears at the end of the paper.

In addition to processor cycle count, we also present PA-RISC and IA-64 code sizes, register usage, and instruction-level parallelism. Finally, we describe the programming approaches we employed for effective use of both architectures. We would be happy to share full details with the finalists' authors under non-disclosure terms.

Methodology

We focused on hand-optimized assembly language implementations of the algorithms for 128-bit keys and 128-bit blocks, using compiled codes as sanity checkers. We agree with Bruce Schneier that AES codes will be implemented in this manner in actual systems; this also leads to the clearest comparisons between instruction set architectures. Codes for this study were optimized for performance, not code size or table size.

For PA-RISC we measured execution speeds on a PA-8500. We timed executions using the PA-RISC 64-bit interval timer, which counts actual clock cycles. To eliminate cache and system effects, we ran tens of millions of executions, varying keys and data blocks on a lightly loaded system, and profiled those runs with minimum cycle counts. We observed that runs often would differ by only a few cycles, and that the cycle counts formed Gaussian distributions. It was further observed that the input value (input key for keying, data block for encryption/decryption) noticeably affected performance for algorithms that used table look-ups. Thus, while the PA-RISC times are best observed times, we also show the distribution's average and maximum values.

Lacking IA-64 hardware, we employed three different types of simulators. Initial debugging used a fairly fast and purely functional instruction set simulator. The second type was considerably slower, but simulated parallel execution, latencies, and memory hierarchy behavior. This was used for additional code validation and preliminary execution cycle counts.

These simulators, while useful, did not guarantee absolute fidelity to the chip designs. Therefore, final timings used fully simulated RTL designs of the Merced (now Itanium) and McKinley chips. This approach was extremely slow, and our results often varied from day to day, as engineers improved their designs. We constructed special tools that automatically prepared test inputs and displayed the cycle-by-cycle behavior of the microprocessor pipeline. The memory hierarchy was initialized for each run, and the timing could be computed by subtracting cycle numbers from the pipeline output.

Notation

`A <<< n`	Left rotation by `n` bits
`A >>> n`	Right rotation by `n` bits
`A ⊕ B`	Bit-wise Exclusive-OR
`A +.× B`	Matrix multiplication
`[b0, b1, ..., bn]`	Column vector, LSB first

PA-RISC Facts

PA-RISC first shipped in 1986 and is the processor for Hewlett-Packard's RISC workstation and server products. Architecture features include 64-bit virtual addressing, 32 general-purpose registers, and 32 floating point registers. Current processors implement the 64-bit Version 2.0 of the PA-RISC architecture.

[1] John S. Worley **jworley@fc.hp.com** William S. Worley, Jr. **worley@hpl.hp.com**
Tom W. Christian **twc@fc.hp.com** Christopher S. Worley **cworley@fc.hp.com**

This study utilized the PA-8200 and PA-8500 microprocessor chips. Both of these chips are out-of-order superscalar designs, capable of executing two memory operations and two integer or floating point instructions per cycle. Only one store instruction can complete per cycle. Careful software scheduling is required to realize the full parallelism.

IA-64 Overview

This section provides a *very* brief overview of the IA-64, highlighting features in the discussions that follow. Readers familiar with the architecture can skip this section.

Parallelism and Functional Units

The majority of processor architectures specify sequential instruction execution. Microarchitectures then employ superscalar logic to issue multiple instructions in parallel whenever possible. In contrast, the IA-64 architecture puts all the parallelism cards on the table. There are four types of functional units: **M** (memory), **I** (integer), **F** (floating point), and **B** (branch); each IA-64 implementation has two or more of each of these units. IA-64 hardware detects when program parallelism exceeds the capabilities of the implementation, but responsibility for organizing instructions to execute in parallel is wholly with the programmer or compiler.

Instructions, Bundles, and Issue Groups

There is a corresponding instruction class for each functional unit type, although a specific instruction may not be able to execute on all units of that type in a given implementation. In addition, there is an **A** (ALU) instruction class that can execute on both **I** and **M** units. **A** instructions include most integer arithmetic and logical operations, so that otherwise idle memory units can be used for parallel computation.

Three instructions are grouped into a *bundle*, where all instructions in the bundle may be eligible to be issued in parallel to functional units specified by the bundle type. Sequential bundles that can issue in parallel form an *issue group*. One characteristic of an IA-64 implementation is the maximum number of bundles that can issue together. For example, a processor that can issue at most two bundles in one cycle is referred to as a “two-banger.”[2]

Registers and the Register Stack

IA-64 provides 128 64-bit integer registers. The low 32 registers (`r0` - `r31`) are common for all code. For function arguments and local values, each procedure can allocate up to 96 additional registers in a *register stack frame*. Saving and restoring registers in the register stack is handled by an independent hardware thread, so that no registers need to be saved and restored explicitly.

In addition to the integer registers, IA-64 provides 128 extended precision (64-bit mantissa, 17-bit exponent) floating point registers, 64 1-bit predicate registers (see below), and eight branch registers for indirect branches.

Predication

A powerful feature of IA-64 is *instruction predication*. Every instruction, except for certain branch and control instructions, is predicated, i.e., its execution is enabled or disabled by one of the 64 predicate bits. One predicate, `p0`, is hardwired to ‘1’ for instructions that execute unconditionally or cannot be predicated. Predicates are set or cleared by compare instructions and certain floating-point instructions. Also, the 64 predicates can be read or set in parallel using special instructions. Predication allows, for example, one of two instructions to execute based on a comparison condition, or for instructions to be enabled during the first pass of a loop and disabled for all subsequent iterations.

Counted Loops

IA-64 provides hardware support for counted loops. The special registers `ar.lc` (loop counter) and `ar.ec` (epilogue counter) control when the branch instructions `br.ctop` and `br.cexit` are taken. For example, if `ar.lc` is set to 9 and `ar.ec` is set to 0, a counted loop will execute 10 times if the loop ends with `br.ctop`, 9 times if the loop begins with `br.cexit`. The hardware is designed to predict perfectly when a branch will be taken or fall through, so that counted loops can execute with no branch penalties.

Rotating Registers

When a subroutine allocates a register stack frame, some or all of the local registers, starting from `r32`, can be set to *rotate*. Each time a counted loop branch is taken, the rotating registers are circularly renamed such that the next iteration of the loop can operate on different data without changing the register name. For example, if there are eight registers designated as rotating, the renaming is as follows:

$$r32 \rightarrow r33 \rightarrow r34 \rightarrow r35 \rightarrow r36 \rightarrow r37 \rightarrow r38 \rightarrow r39 \rightarrow r32$$

Fixed portions of the floating point and predicate registers also rotate. The high 96 floating point registers (`f32` through `f127`) rotate. The high 48 predicate registers (`p16` to `p63`) also rotate, but with a slight difference. While the loop counter `ar.lc` is non-zero, a ‘1’ value is shifted into `p16`; if `ar.lc` is zero and the epilogue counter `ar.ec` > 1, a ‘0’ value is shifted in instead.

Programming Issues

There are three operations commonly used in cryptographic algorithms that are not fully realized in the integer hardware on PA-RISC and IA-64: fixed 32-bit rotations, variable 32-bit rotations, and 32x32→32 unsigned integer multiplies.

[2] This term comes from the slang term for a two-cylinder engine. While three-banger or more implementations are foreseeable, it seems unlikely that IA-64 will ever give rise to, say, a V12.

PA-RISC

On PA-RISC, fixed rotations can be executed in one cycle using the shift right pair word (`shrpw`) instruction. This instruction concatenates the low 31 and 32 bits from left and right source registers, respectively, shifts right the specified distance, and leaves the high 32 bits undefined. If the two source registers are the same, the low bits are concatenated with the high bits, exactly as would occur in a rotation. Thus, fixed rotations on PA-RISC can be defined as follows:

```
ROTR   .macro       src, dst, count
       shrpw        src, src, count, dst
       .endm
ROTL   .macro       src, dst, count
       shrpw        src, src, 32 - count, dst
       .endm
```

Variable rotations use the same strategy, except that an extra cycle is required to move the shift distance into the SAR (shift amount register). For a right rotation, the actual shift distance is used. For a left rotation, the 5-bit complement of the distance is used and the value is pair-shifted right one before the variable shift. The left shift also executes in two cycles since the `mtsarcm` (move to SAR complement) and the first `shrpw` can issue in the same cycle on the PA-8000 family.

Integer multiplication on PA-RISC requires using the unsigned integer multiply in the floating point unit. Since the only path for moving data between the integer and floating point units is memory, the multiplicands must be stored, loaded into the FPU, multiplied, stored again, and reloaded into the integer unit. This adds latencies on both sides of the multiply, in addition to the multiply time itself.

IA-64

Although the IA-64 architecture has a shift right register pair instruction, it only operates on full 64-bit registers. This can still be used to implement 32-bit fixed rotations in two cycles as follows:

```
dep.z  TMP = src, 32, 32
shrp   dst = src, TMP, count + 32
```

for right rotations, and

```
dep.z  TMP = src, 32, 32
shrp   dst = src, TMP, 64 - count
```

for left rotations.

The `dep.z` instruction puts the low 32 bits of the source register in the high half of a temporary register, clearing the low half. The pair-shift concatenates the low bits with the high bits and shifts far enough to put the proper set of bits in the low half of the destination. Like the PA-RISC instruction, the destination's high half is not cleared. None of the AES finalists require these bits to be cleared; however, the `zxt4` instruction can be used if necessary.

On IA-64, variable rotates are implemented much as in the C language: shift left `j`, shift right `(32 - j)`, `OR` or `ADD` the results together. This involves four operations and a minimum of three cycles. The variable shifts are executed on the multimedia units (MMUs).

Like PA-RISC, the IA-64 primary integer multiply is implemented on the floating point unit and involves latency cycles to move back and forth. However, 16x16 MMU multiplies and parallel adds can be used to compute and sum the partial products instead. This is effective when only the low 32 bits of the result are of interest. In particular, the parallel 16-bit unsigned multiply and shift instruction (`pmpyshr.u`) can be used to complete a 32x32→32 multiply.

If we consider multiplicands derived from `A` as four 16-bit elements, `A`^{2} can be computed with two multiplies and two adds as follows:

0	0	A_{HI}	A_{LO}	***>>0**	0	0	A_{LO}	A_{LO}	=	0	0	$A_{HI}A_{LO}$<15..0>	A_{LO}^2<15..0>
												+	
0	0	A_{LO}	0	***>>16**	0	0	A_{LO}	A_{LO}	=	0	0	A_{LO}^2<31..16>	0
												+	
										0	0	$A_{HI}A_{LO}$<15..0>	0

One of the operands is just the argument, `A`. The other two arguments are generated by the 16-bit `mux` MMU instructions; the additional addend is derived from the first product using the 16-bit `mix` instruction. The general 32x32→32 requires three multiplies and two additions. If we consider multiplicands derived from `A` and `B` as four 16-bit elements, the operations are:

0	0	A_{HI}	A_{LO}	***>>0**	0	0	B_{LO}	B_{LO}	=	0	0	$A_{HI}B_{LO}$<15..0>	$A_{LO}B_{LO}$<15..0>
												+	
0	0	A_{LO}	0	***>>0**	0	0	B_{HI}	B_{LO}	=	0	0	$A_{LO}B_{HI}$<15..0>	0
												+	
0	0	A_{LO}	0	***>>16**	0	0	B_{LO}	B_{LO}	=	0	0	$A_{LO}B_{LO}$<31..16>	0

Two of the operands are just the arguments, `A` and `B`. The other two arguments are generated by the 16-bit `mix` and `mux` MMU instructions.

It has been noted that with better hardware support for 32-bit rotations and 32x32→32 multiplication, all the AES finalists will outperform Pentium on IA-64. In the performance analysis for each algorithm, we have estimated performance for a hypothetical IA-64 implementation, called IA-64++, with the following enhancements:

- A single-cycle shift right pair word instruction, as in PA-RISC
- Single-cycle, 32-bit, left and right variable rotate instructions
- A two-cycle 32x32→32 unsigned multiply

Mars

The Mars encryption scheme (IBM team) uses a mix of approaches: substitution boxes, Feistel networks, multiplication, and fixed and variable rotates. The single substitution box, `S[]`, is fixed, and is employed both as a 512 word array (9-bit index), and as low (`S0[]`) and high (`S1[]`) 256 word arrays (8-bit index). The principal challenges for PA-RISC and IA-64 implementations are the 32x32→32 multiply and variable rotates.

Keying[3]

Mars keying initializes the first `N` elements of a fifteen-element array, `T[]`, to the input key *`k`*`[]`, where N is the size of the key in 32-bit words. The key is then padded to 15 words by setting `T[N]`←`N` and zeroing the remainder of the array. Instead of generating the entire expanded key directly, Mars generates ¼ of the array, or 10 words, each time, repeating the process four times to develop the entire key array, `K[]`. There are three steps in each iteration: linear transform, stirring, and storing. The linear transform applies the formula:

```
T[i] = T[i] ⊕ ((T[i-7 mod 15] ⊕ T[i-2 mod 15]) <<< 3) ⊕ (4i + R)
```

to each element of the array, where `R` is the iteration count (0..3). Stirring uses the following formula:

```
T[i] = (T[i] + S[T[i-1 mod 15] & 0x1ff]]) <<< 9
```

applied to each word, repeated four times. Finally, 10 words from the intermediate array are stored in the expanded key array as follows:

```
K[10×R + i] = T[4i mod 15]
```

which effectively stores words 0, 4, 8, 12, 1, 5, 9, 13, 2, and 6, in that order, from the temporary array. After all the expanded key words are generated, those used in multiplication (`K[5]`, `K[7]`, …, `K[35]`) are modified if they are weak, i.e., contain long runs of 1’s or 0’s. The algorithm for identifying weak key words comes from the Mars implementation by Brian Gladman.

PA-RISC

The PA-RISC implementation keeps `T[]` in registers. The linear transform, the inner stirring loop, and key stores are straight-lined. In the fix-up phase, the two-ALU PA-RISC has sufficient execution bandwidth to compute the fix-up mask in parallel with looking for long runs of 1’s or 0’s. If there are no such runs, the remainder of the fix-up is skipped. Using the authors' estimates that statistically 1 out of 41 keys are weak, the extra computation is skipped 97.6% of the time, a performance win even with a branch penalty.

IA-64

The IA-64 Mars keying implementation uses software pipelining to increase keying speed. The routine allocates a 16-register stack frame, all of which are rotating. The register usage is as follows (indices are modulo 15):

r32	r33	r34	r35	r36	r37	r38	r39	r40	r41	r42	r43	r44	r45	r45	r47
T_{i-1}	T_{i-2}	T_{i-3}	T_{i-4}	T_{i-5}	T_{i-6}	T_{i-7}	T_{i-8}	T_{i-9}	T_{i-10}	T_{i-11}	T_{i-12}	T_{i-13}	T_{i-14}	T_i	T_x

By assigning $T_x \leftarrow T_i$ at the end of the loop, this organization implements a 15-register rotation. The linear transform XORs T_{i-2} (`r33`) and T_{i-7} (`r38`), rotates the result, the XORs with T_i and the iteration constant (`4i + R`). This would normally require four cycles; however, the transform can be reorganized into a two-stage, two-cycle pipeline. The first stage computes $T_{i-2} \oplus T_{i-7}$ and extracts the high three bits of the result; the second phase computes $T_i \oplus$ (`4i + R`) and completes the rotation, then XORs the final result. The loop uses rotating predicates to disable the second phase on the first iteration, while the last execution of the second phase is handled after the loop so that the values return to their initial positions when the loop is complete.

Pipelining the inner stirring loop is limited by the use of `T[i-1 mod 15]` in computing `T[i]`; however, the high-order nine bits extracted for rotation can be used to start the S-Box look-up for the next iteration. This allows a two-stage, four-cycle pipeline, which executes 33% faster than the 6-cycle, non-pipelined equivalent.

Like PA-RISC, the fix-up mask can be computed in parallel with looking for runs of 1’s and 0’s. Unlike PA-RISC, branches include ‘hints’, so that the branch penalty is only incurred for weak keys, or 2.4% of the time.

Encryption

Mars encryption consists of four phases, each repeated eight times: forward mix, forward keyed transform, backward keyed transform, and backward mix. The forward and backward mixing uses table look-ups, fixed rotation, XORs, and addition and subtraction in a rotating pattern, e.g., `fmix(A, B, C, D)`, `fmix(B, C, D, A)`, etc. There are asymmetric additions in steps 1, 2, 4, and 5 of the forward mix, with corresponding subtractions in steps 2, 3, 5 and 6 of the backward mix.

[3] This is the ‘tweaked’ version of the Mars keying. The implementation of the initialization, mixing, and stirring phases of the original scheme is discussed in Appendix B. The key fix-up is identical for both schemes.

The core of the keyed transforms is the *E* function, which takes one data word and uses table look-up, multiplication, variable rotation, additions and XORs to generate three data words (`L`, `M` and `R`) to add or XOR with the other three data words as follows:

```
 Forward Mode         Backward Mode
D[1] += L            D[1] ^= R
D[2] += M            D[2] += M
D[3] ^= R            D[3] += L
```

On PA-RISC, the mixing phases are coded as straight-line operations. Even with the four table look-ups per step, there is enough memory bandwidth to load the 16 multiplicative keys into the floating-point unit at the same time. The real bottleneck is the integer multiply in the *E* function: the data word must be rotated, stored, loaded into the floating point unit, multiplied, stored again and reloaded into an integer register. Although an addition and table look-up can be evaluated in parallel, these do not fully amortize the performance cost of the multiply.

On IA-64, both forward and backward mixing can be coded as a single loop: the asymmetric operations are controlled by loading a specific bit pattern in the rotating predicates, enabling the appropriate operation at the proper step. Because of perfect branch prediction with counted loops, this approach executes in the same cycle count as straight-line code.

On IA-64, MMU multiplies are used to compute the *E* function multiplication. Once the multiplication is complete, the remainder of the *E* function can be evaluated. Like the mixing phases, a predetermined bit pattern loaded in the rotating predicates controls whether the forward or backward mode operations are enabled at each step.

Performance

Cycles	**Pentium**	**PA-RISC**			**IA-64**	**IA-64++**
		Min	Average	Max		
Keying	2128	1797	1804.65	1879	1408	1408
Keying (Original)	3894	1969	1975.89	2060	1903	1313
Encryption	320	540	563.01	584	511	255
Decryption	374	538	552.37	566	527	271

On PA-RISC, Mars keying executes in 1797 cycles, compared to the best-reported Pentium results of 2128, a 15.6% performance advantage. Encryption and decryption, however, run slower due to the multiplication overhead: 68.8% slower for encryption (540 vs. 320) and 43.9% slower for decryption (538 vs. 374).

On IA-64, keying completes in 1408 cycles, a 33.8% performance gain. Encryption and decryption, with the extra cycles required for multiplication and variable rotation, are slower than Pentium: 59.7% slower for encryption (511 vs. 320) and 40.9% slower for decryption (527 vs. 374). Keying on IA-64++ is the same 1408 because the software pipelines hide the extra cycles needed for rotation. Encryption improves to 255 cycles (20.3% faster than Pentium), and decryption also improves to 271 cycles (27.5% faster).

RC6

The principal programming challenge when implementing RC6 (Rivest, Robshaw, Sidney, Yin) on PA-RISC and IA-64 is the lack of the fast 32x32→32 multiply and variable rotate primitive the algorithm requires for performance. On the positive side, IA-64's rotating integer registers and instruction predication simplify data management and allow for a very compact code size.

Keying

RC6 keying starts with the input key, `L[]`.The key array, `S[]`, is initialized using the two magic numbers P_{32} = 0xB7E15163 and Q_{32} = 0x9E3779B9, as follows:

$$S[0] = P_{32}$$
$$S[1] = P_{32} + Q_{32}$$
$$S[2] = P_{32} + 2 * Q_{32}$$
$$S[3] = P_{32} + 3 * Q_{32}$$
$$. \; . \; .$$

The keying algorithm then performs three mixing passes over the two arrays:

```
A = S[i] = (S[i] + A + B) <<< 3
B = L[j] = (L[j] + A + B) <<< (A + B)
```

where A and B are initially zero, and i and j count circularly through the key and input key arrays, respectively. If the first pass through the key array is handled separately, it is possible to combine the key array initialization with the first mixing phase. The first mix can also be partially hard coded, since A = B = 0, and S[0] = P_{32}. Since, after the first loop pass, B is just the previous, modified input key word, the variable B is replaced with LPREV(k), the user input key L[(k-1) mod 4]. The first pass is coded as follows:

```
keyVal = P32;
A = T = ROTL(P32, 3);
for (k = 1; k < NKEYS; ++k) {
      LPREV(k) = ROTL(LPREV(k) + T, T);
      keyVal += Q32;
      S[k - 1] = A;
      A = ROTL(keyVal + A + LPREV(k), 3);
      T = LPREV(k) + A;
}
S[NKEYS - 1] = A;
```

This organization saves one full load and store of the key array and does not require computing the modulus 2*r + 4, where r is the number of encryption rounds. The last two passes are identical, with a similar structure to the first pass, but do not, of course, re-initialize the key array.

For PA-RISC, each instance of the loop can be unrolled four ways, with the input key words reordered circularly each time - this eliminates loading and storing the keys, and the modulus computation on the input key index.

The IA-64 architecture suggests a different strategy for implementation. The large register file allows *the entire key array* to be kept in registers; the rotating integer registers naturally mimic the way data flows through the computation, such that no indexing or modulo operations are required. The keying routine allocates a 56-register stack frame, all of which are rotating. The rotating registers are allocated as follows:

r32-r33	r34	r35	r36	r37	r38	r39	r40	r41	r42-r82	r83	r84-r87
Unused	L_X	L_n	L_{n+1}	L_{n+2}	L_{n+3}	S_X	S_{Active}	S_{Prev}	Key Array	S_{Next}	Unused

where <L_n … L_{n+3}> are initialized from the user input key. In order to circulate the keys and key array separately, $L_X \leftarrow L_{n+3}$ and $S_X \leftarrow S_{Next}$ before the registers are rotated. Each time through the loop, the code operates on L_n, S_{Active}, and S_{Prev}. Rewriting the mixing loop in these terms:

```
for (k = 1; k < NKEYS; ++k) {
      L_n = ROTL(L_n + T, T);
      A = ROTL(S_Active + S_Prev + L_n, 3);
      T = L_n + A;
      S_Active = A;
      L_X  = L_n+3;
      S_X  = S_Next;
}
```

Predicated instructions enable key array initialization during the first mixing pass and storing the final key words during the final pass, all within the same code loop and *without* branching. There are enough unused instruction slots to compute the two qualifying predicates with no additional cycles. The keying routine is thus coded in a single loop:

```
for (k = 1; k < 3 * NKEYS; ++k) {
      L_n = ROTL(L_n + T, T);      if (firstMix) S_Active = S_Prev + Q_32;
      firstMix = k < NKEYS-1;    lastMix = k >= 2 * NKEYS;
      S_Prev = A;                if (lastMix) *S++ = A;
      A = ROTL(S_Active + A + L_n, 3);
      T = L_n + A;
      L_X  = L_n+3;
      S_X  = S_Next;
}
*S = A;
```

This coding is extremely compact: the entire routine consists of 39 instructions in 16 IA-64 bundles; the core loop is 20 instructions.

Encryption

The RC6 definition is compact and elegant, but the algorithm relies on a fast 32x32→32 multiply and variable rotate for performance. To multiply on PA-RISC, the two data words must be stored, loaded into the floating point unit, multiplied, stored again and reloaded into integer registers. The inner loop is unrolled to rotate the data words.

On IA-64, MMU multiplies are used to compute A^2. Once the full multiplication is complete, the shladd instruction computes the final product $2A^2 + A \equiv A*(2A + 1)$. Using rotating registers for the data words, RC6 encryption can be coded in a single loop.

Performance

Cycles	**Pentium**	**PA-RISC**			**IA-64**	**IA-64++**
		Min	Average	Max		
Keying	1632	1077	1077	1077	1581	1057
Encryption	243	580	590.76	597	490	150
Decryption	226	493	496.37	499	490	130

On PA-RISC, RC6 keying executes in 1077 cycles, compared to the best-reported Pentium results of 1632, a 34% performance advantage. Encryption and decryption, however, run slower due to the multiplication overhead: 138% slower for encryption (580 vs. 243) and 118% slower for decryption (493 vs. 226).

On IA-64, keying completes in 1581 cycles, a 3.1% performance gain. Encryption and decryption, with the extra cycles required for multiplication and variable rotation, are slower than Pentium: 101.7% slower for encryption (490 vs. 243) and 116.8% slower for decryption (490 vs. 226). For IA-64++, keying is estimated to run in 1057 cycles, 54% faster than Pentium, encryption in 150 cycles (38.3% faster), and decryption in 130 cycles (42.5% faster)

Rijndael

The principles for a fast Rijndael (Daemen, Rijmen) implementation are largely explained in the algorithm specification. A short comment in section 5.2.2 summarizes the general approach:

> *"In the table-lookup implementation, all table lookups can in principle be done in parallel. The EXORs can be done in parallel for the most part also."*

This turns out to be an understatement. In other AES candidates, parallelism must be squeezed from the specification, while Rijndael's parallelism cup runneth over. Even the keying phase has considerable parallelism, as will be shown.

Realizing this parallelism requires five 4K tables, as discussed below, although only two tables are used for any one operation. Each 4K table is made up of 4 256x4 byte tables, where each 1K table is rotated one byte position from the previous. The tables and the operations they're used in are:

S-Box	**Keying, Encryption** Implements byte substitution only
I-Box	**Decryption** Implements inverse byte substitution only
Column Mix	**Encryption** Main substitution box - combines the byte substitution and column mix operations
Inverse Mix	**Decryption** Inverse substitution box - combines the byte substitution and inverse column mix operations
Key Mix	**Keying** Column mix box for computing the inverse key table

These tables are all derived from the basic GF(2^8) mathematics outlined in the specification. A simple C program is used to generate all tables and print them as C array declarations to compile and link with the algorithm codes. While 20K bytes of tables may be not optimal for some target implementations, large memory, large cache machines like PA-RISC and IA-64 gain substantial performance with what is negligible extra data. Rijndael outperforms all other AES submissions in keying, encryption, and decryption. In particular, Rijndael keying is a full order of magnitude faster than most other algorithms.

Keying

Rijndael key expansion looks largely serial. There are four look-ups every fourth key word, but little else to suggest parallelism. The discussion in section 5.3.3, however, shows that decryption can be more efficiently implemented if an "inverse" key table is used. If the basic key generation loop is unrolled four times, we can combine the inverse key computation with the key generation:

```
A = SubByte(RotByte(D)) ^ Rcon[i];
B = B ^ A;
C = C ^ B;
D = D ^ C;
IA = InvMixColumn(A);
IB = InvMixColumn(B);
IC = InvMixColumn(C);
ID = InvMixColumn(D);
```

Clearly, the `InvMixColumn` operation, which is four byte-indexed lookups into four 256-entry tables and three XORs, can begin as soon as the key word is ready. Thus, both the forward and inverse key tables can be computed in the same time as computing the inverse table. As a minor space optimization, the last forward key and first inverse key, which are identical, are stored only once in a combined key table.

Both the PA-RISC and IA-64 implementations are straightforward: as soon as the forward key is available, start the look-ups for the inverse key. Two look-ups are performed on the key word `A`, but only one set of byte extractions is needed, saving four operations per round. PA-RISC has 28 registers available to a subroutine: all of these are needed to hold the intermediate results. The large register file on IA-64 provides enough temporary registers to perform the computation with maximum concurrency. Rijndael keying improves greatly when everything can be kept in registers.

On PA-RISC, a load address can be the sum of a base register and a scaled offset register; thus, table look-up requires two instructions. IA-64, however, only takes a load address from a register without offset. Therefore, a table look-up must explicitly scale the index and add it to the desired table address: this is accomplished with the `shladd` instruction. The sequence of extract, scale and add, load is pipelined, so that the entire look-up sequence only requires one extra cycle over the equivalent PA-RISC sequence. The greater parallelism in IA-64 allows the forward key computation and XOR trees to overlap the look-ups, giving it an overall performance advantage.

Encryption

Rijndael encryption, while defined as several, separate steps, can be collapsed into a single set of table look-ups by (1) computing the look-up tables to combine the byte substitution and column mix operations, and (2) selecting the index bytes from the data block to reflect the row rotation in each round. Decryption is identical except for the look-up table and the order of byte selection. It is not surprising, then, that encryption and decryption are very similar to keying, except that only 16 look-ups are done per round instead of the 20 performed for each keying round.

Performance

Cycles	**Pentium**	**PA-RISC**			**IA-64**	**IA-64++**
		Min	Average	Max		
Keying	1338	239	249.25	261	148	148
Forward Keying	217	85	92.18	101	104	104
Encryption	284	168	175.5	193	124	124
Decryption	283	168	175.88	192	125	125

On PA-RISC, Rijndael full keying executes in 239 cycles, compared to the best-reported Pentium results of 1338, a 5.6:1 performance advantage. Encryption and decryption are faster: 40.9% faster for encryption (168 vs. 284) and 40.6% faster for decryption (168 vs. 283). On IA-64, keying completes in 148 cycles, a 9:1 performance improvement over Pentium. Encryption and decryption are also faster: 56.3% faster for encryption (124 vs. 284) and 55.8% faster for decryption (125 vs. 283).

The parallelism of Rijndael saturates a two-banger IA-64. To explore the limits of Rijndael's parallelism, a code schedule was developed for a hypothetical, four-banger implementation. With this 12-way parallel IA-64, the inner loop of Rijndael encryption can be executed in 7 cycles, which suggests a total encryption time of 74 cycles per 128-bit data block. This is only one cycle short of the theoretical limit of 6 cycles per round for an arbitrarily wide IA-64 implementation, which would perform 16 extracts, 20 address computations, 20 loads, then three levels of XORs.

Serpent

The heart of the Serpent algorithm (Anderson, Biham, Knudsen) is the set of Boolean equations implementing the "bit-slice" substitution boxes. One set of equations was submitted with the AES proposal; Brian Gladman and Sam Simpson used a recursive expression search program to develop an alternative set of equations that improved performance on the Pentium-II platform. Dr. Gladman, however, cautions on his Serpent web page[4]:

> *"On any particular machine it will be desirable to experiment with the order of terms (where there is quite a lot of flexibility) and with the reuse of the temporary variables used during function evaluation."*

Taking this advice to heart, the two sets of equations, along with an earlier version of Gladman's equations, and a set of equations optimized for Pentium submitted to the authors by Dag Arne Osvik[5], were analyzed according to the following metrics:

Ops Count of Boolean operations required to compute the substitution or reverse substitution function. The equation parser looks for occurrences of `A & ~B` to take advantage of the `and-complement` instruction in both the PA-RISC and IA-64 instruction sets.

Cycles Number of steps required to complete the computation on a highly parallel machine, such as IA-64, and a two-ALU operation superscalar machine, such as PA-RISC.

Width For IA-64, the largest number of operations executed concurrently.

Temps Number of temporary values. In order to reduce the number of temporaries, a simple register analysis was performed that first re-used the output terms as intermediate results, then assigned temporaries as needed by the computation.

The results of this analysis for IA-64, summarized in Table 1 below, are interesting: even though the Gladman equations consistently have fewer operations than the others, only 4 of the 16 sets compute faster. When the equations are analyzed for two-ALU

[4] The expression search program, Boolean equations and reference implementations are available at **http://www.btinternet/~brian.gladman/cryptography_technology/Serpent**

[5] Dag Arne Osvik **osvik@ii.uib.no**

operation on PA-RISC, the results (Table 2) favor Gladman's equations, but four of Osvik's equations compute faster. A follow-up submission from Mr. Osvik for S-Box 3 resulted in a spectacular, 4-cycle, solution for IA-64, even though it has the highest operation count of any equation.

The conclusion here is that there is no optimal set of bit-slice equations for all Serpent implementations: the capability and constraints of the target machine must be carefully considered. The authors invite others to submit their own equations for analysis, and offer the analysis tools used here to the Serpent team for their own use.

Keying

Serpent keying starts with the input key, padded to 256 bits, and generates 132 4-byte values with the recurrence:

$$W_i = (W_{i-8} \oplus W_{i-5} \oplus W_{i-3} \oplus W_{i-1} \oplus \Phi \oplus i) \lll 11$$

where W_{-8} = input key word 0, W_{-7} = input key word 1, etc., and Φ is 0x9e3779b9, derived from the Golden ratio. The resulting values, $[W_0 \ldots W_{131}]$, are then processed in groups of four, $<W_n, W_{n+1}, W_{n+2}, W_{n+3}>$, applying the Serpent forward substitution boxes in the order S_3, S_2, S_1, S_0, S_7, …, S_4, S_3. This generates the 33 128-bit keys required for encryption.

Inspecting the recurrence, there is an active state of eight words and that W_i replaces W_{i-8} at each step. If we label the initial key words $W_{-8} = A$, $W_{-7} = B$, … $W_{-1} = H$, we can rewrite the recurrence as the following pattern:

$$\begin{aligned}
A' &= (A \oplus D \oplus F \oplus H \oplus \Phi \oplus 0) \lll 11 \\
B' &= (B \oplus E \oplus G \oplus A' \oplus \Phi \oplus 1) \lll 11 \\
C' &= (C \oplus F \oplus H \oplus B' \oplus \Phi \oplus 2) \lll 11 \\
D' &= (D \oplus G \oplus A' \oplus C' \oplus \Phi \oplus 3) \lll 11 \\
E' &= (E \oplus H \oplus B' \oplus D' \oplus \Phi \oplus 5) \lll 11 \\
F' &= (F \oplus A' \oplus C' \oplus E' \oplus \Phi \oplus 6) \lll 11 \\
G' &= (G \oplus B' \oplus D' \oplus F' \oplus \Phi \oplus 7) \lll 11 \\
H' &= (H \oplus C' \oplus E' \oplus G' \oplus \Phi \oplus 8) \lll 11 \\
&\quad \dots \\
A' &= (A \oplus D \oplus F \oplus H \oplus \Phi \oplus 128) \lll 11 \\
B' &= (B \oplus E \oplus G \oplus A' \oplus \Phi \oplus 129) \lll 11 \\
C' &= (C \oplus F \oplus H \oplus B' \oplus \Phi \oplus 130) \lll 11 \\
D' &= (D \oplus G \oplus A' \oplus C' \oplus \Phi \oplus 131) \lll 11
\end{aligned}$$

This formulation has some limited parallelism in the XOR trees. Eventually, the equations will serialize on the 11-bit rotation, but the overall sequence can be organized on a parallel machine to minimize the performance effect. Intermediate loads and stores can be eliminated by overlapping the S-box lookup for $<W_n, W_{n+1}, W_{n+2}, W_{n+3}>$ with the computation of $<W_{n+4}, W_{n+5}, W_{n+6}, W_{n+7}>$. Because different S-boxes are used at each step, the highest performance for Serpent keying is realized by a straight-line implementation.

On PA-RISC, limited to two-way integer instruction parallelism, each set of four recurrence computations saturates the processor for 11 cycles (22 operations). The 11-bit rotation is implemented with a single instruction (shrpw); common subexpressions (e.g., $F \oplus H$) remove two of the 24 operations (five XORs and one rotate per step, times four steps). Since PA-RISC does not have an immediate XOR operation, the $(\Phi \oplus i)$ term is computed by adding the low 11 bits of the value (constant for each step) to the high 21 bits (constant for all steps); thus, the computation still occurs in one cycle. To avoid errors, the 11-bit values are generated by a simple program.

IA-64 rotation requires two instructions (deposit and shift register pair). This increases the cycle count for computing four steps from 11 on PA-RISC to 14. However, the machine's greater parallelism can be employed to overlap S-Box and recurrence logic as follows:

$$\begin{array}{ll}
\mathrm{Recurrence}(W_0, W_1, W_2, W_3) & \\
\mathrm{Recurrence}(W_4, W_5, W_6, W_7) & \mathrm{Sbox3}(W_0, W_1, W_2, W_3) \\
\mathrm{Recurrence}(W_8, W_9, W_{10}, W_{11}) & \mathrm{Sbox2}(W_4, W_5, W_6, W_7) \\
\mathrm{Recurrence}(W_{12}, W_{13}, W_{14}, W_{15}) & \mathrm{Sbox1}(W_8, W_9, W_{10}, W_{11}) \\
\qquad\qquad \dots & \qquad\qquad \dots \\
\mathrm{Recurrence}(W_{124}, W_{125}, W_{126}, W_{127}) & \mathrm{Sbox5}(W_{120}, W_{121}, W_{122}, W_{123}) \\
\mathrm{Recurrence}(W_{128}, W_{129}, W_{130}, W_{131}) & \mathrm{Sbox4}(W_{124}, W_{125}, W_{126}, W_{127}) \\
 & \mathrm{Sbox3}(W_{128}, W_{129}, W_{130}, W_{131})
\end{array}$$

Each step in this parallel evaluation, including storing the key words, executes in the 14 cycles needed for the recurrence alone, yielding a substantial speed-up for Serpent keying.

Encryption

Serpent encryption and decryption use 32 rounds of key exclusive OR's, substitution box logic and linear transforms. The S-box issues are almost identical to those for keying, as discussed above. The linear transform, which accelerates the avalanche effect, limits the potential for overlap with the S-box computations. Depending on the S-box equations used, at most one or two cycles can be removed per S-box; the current implementation overlaps one cycle for six of the eight S-box equations.

The forward linear transform, diagrammed in Figure 1, consists of 16 operations (six fixed rotations, two rotations, eight exclusive-OR's). Ideally, this sequence can be executed in seven cycles on a parallel machine:

$$\begin{array}{lll}
X_0 = X_0 \lll 13 & X_2 = X_2 \lll 3 & \\
X_1 = X_1 \oplus X_0 & X_3 = X_3 \oplus X_2 & T1 = X_0 \ll 3 \\
X_1 = X_1 \oplus X_2 & X_3 = X_3 \oplus T1 & \\
X_1 = X_1 \lll 1 & X_3 = X_3 \lll 7 & \\
X_0 = X_0 \oplus X_3 & X_2 = X_2 \oplus X_3 & T2 = X_1 \ll 7 \\
X_0 = X_0 \oplus X_1 & X_2 = X_2 \oplus T2 & \\
X_0 = X_0 \lll 5 & X_2 = X_2 \lll 22 &
\end{array}$$

The inverse linear transform, diagrammed in Figure 2, also has 16 operations; however, it can be computed in five cycles:

$$\begin{array}{llll}
X_0 = X_0 \ggg 5 & X_2 = X_2 \ggg 22 & T1 = X_1 \oplus X_3 & T2 = X_3 \ggg 7 \\
X_0 = X_0 \oplus T1 & X_2 = X_2 \oplus X_3 & T3 = X_1 \ll 7 & X_1 = X_1 \ggg 1 \\
X_1 = X_1 \oplus X_0 & X_2 = X_2 \oplus T3 & T4 = X_0 \ll 3 & \\
X_1 = X_1 \oplus X_2 & X_3 = X_3 \oplus X_2 & X_0 = X_0 \ggg 13 & \\
X_3 = X_3 \oplus T4 & X_2 = X_2 \ggg 3 & &
\end{array}$$

On PA-RISC, the single-cycle fixed rotation allows both transforms to execute in eight cycles, optimal for the two-way superscalar machine. The two-cycle rotation on IA-64 increases the operation count to 22, and the dependencies are such that the best implementation for the transforms requires 12 cycles. Loading and XORing the key material in parallel with the transforms can reclaim some performance; however, the linear transformation accounts for over 50% of the encryption and decryption cycles.

As with keying, the best performance is achieved with straight-line code. The program source for both PA-RISC and IA-64 make heavy use of macros and bear strong resemblance to the algorithm specification. An extension of the software tools used to analyze Serpent equations actually produces the raw instruction stream for each equation, in either machine language format, which is then easily integrated into the source program through the macro definitions.

Performance

Cycles	**Pentium**	**PA-RISC**			**IA-64**	**IA-64++**
		Min	Average	Max		
Keying	1292	668	668.79	669	475	380
Encryption	900	580	580	580	565	468
Decryption	885	585	586.62	587	631	407

On PA-RISC, Serpent keying executes in 668 cycles, compared to the best-reported Pentium results of 1292, almost a 2:1 performance advantage. Encryption and decryption also run substantially faster: a 35.6% advantage for encryption (580 vs. 900) and a 33.9% advantage for decryption (585 vs. 885).

On IA-64, the extra parallelism pays off handsomely in keying, where the routine completes in 475 cycles, a 2.7:1 performance gain over Pentium. Encryption and decryption, with the extra cycles required to complete the linear transform, are better than Pentium, although not as overwhelmingly: 37.2% for encryption (565 vs. 900), 28.7% for decryption (631 vs. 885). For IA-64++, keying is estimated to run in 380 cycles, 3.4 times faster than Pentium, encryption in 468 cycles (48.0% faster), and decryption in 407 cycles (54% faster).

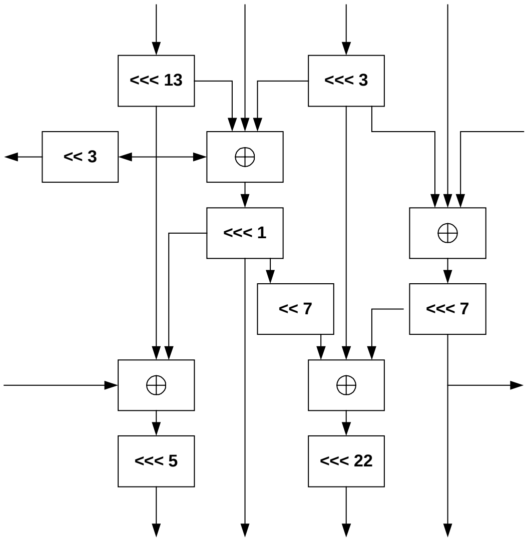


Figure 1 – Serpent Linear Transform

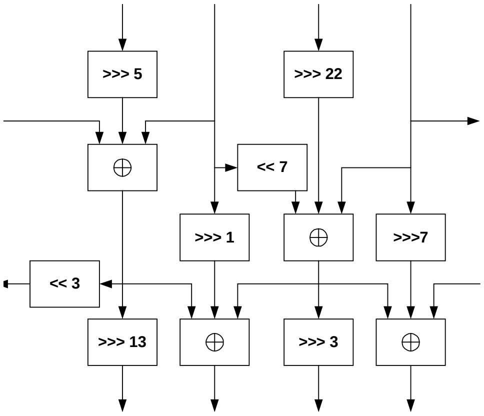


Figure 2 – Serpent Inverse Transform

	AES Submission				Gladman (Best)				Osvik			
	Ops	Cycles	Width	Tmps	Ops	Cycles	Width	Tmps	Ops	Cycles	Width	Tmps
S Box 0	18	9	6	4	15	6	4	3	17	6	4	5
S Box 1	18	9	5	3	14	8	3	2	17	7	3	3
S Box 2	16	9	3	4	16	8	3	3	14	7	3	5
S Box 3	18	7	5	4	16	8	5	3	21	4	6	6
S Box 4	19	7	4	5	15	8	3	3	19	9	3	3
S Box 5	17	8	3	4	16	7	4	3	18	7	3	3
S Box 6	19	6	7	4	17	6	5	4	17	9	3	3
S Box 7	19	8	4	3	17	11	3	3	19	8	4	5
I Box 0	19	8	5	4	15	10	2	2	18	8	3	4
I Box 1	18	9	3	3	17	7	5	2	18	11	3	3
I Box 2	18	7	5	4	16	8	4	3	18	7	3	3
I Box 3	17	7	4	3	17	9	4	4	17	8	3	3
I Box 4	17	7	4	4	17	6	5	5	19	11	3	3
I Box 5	17	7	5	4	16	7	4	3	18	10	2	3
I Box 6	19	6	4	4	17	8	4	2	16	8	3	3
I Box 7	18	9	4	2	17	9	3	2	18	8	4	4

Table 1 - Serpent IA-64 Metrics

	AES Submission			Gladman (Best)			Osvik		
	Ops	Cycles	Tmps	Ops	Cycles	Tmps	Ops	Cycles	Tmps
S Box 0	18	11	3	15	8	2	17	9	2
S Box 1	18	11	3	14	8	2	17	9	3
S Box 2	16	11	4	16	9	3	14	8	3
S Box 3	18	9	4	16	9	3	17	9	3
S Box 4	19	10	6	15	8	3	19	10	1
S Box 5	17	9	4	16	9	3	18	9	1
S Box 6	19	10	4	15	9	3	17	10	2
S Box 7	19	10	3	17	12	5	19	10	2
I Box 0	19	10	4	15	10	2	18	11	2
I Box 1	18	10	3	17	9	3	18	11	2
I Box 2	18	10	3	16	9	2	18	10	2
I Box 3	17	9	3	17	9	4	17	9	1
I Box 4	17	9	5	17	9	4	19	11	2
I Box 5	17	9	4	16	8	4	18	10	2
I Box 6	19	10	5	17	9	3	16	8	2
I Box 7	18	9	3	17	10	2	18	9	2

Table 2 - Serpent PA-RISC Metrics

Twofish

The Twofish block cipher employs a *"Feistel-like structure with additional whitening of the input and output."*[6] The 128-bit plaintext block is split into four 32-bit words. In the input whitening step each 32-bit word is XORed with a different 32-bit input-whitening key. This is followed by 16 rounds in which the left two words are transformed by the F-function. The leftmost word produced by the F-function is XORed with the third word, and the result is rotated to the right by one bit. The rightmost word produced by the F-function is XORed with the fourth word, which previously had been rotated to the left by one bit. For all but the 16th round, the left and right pairs of words then are swapped for the next round. Each of the final four words is XORed with a different 32-bit output-whitening key.

Within the F-function, the first input word is transformed by the g-function. The second input word first is rotated to the left by eight bits, and then transformed by the g-function. The two g-function outputs then are mixed into two new words by a Pseudo-Hadamard Transform (PHT). After mixing, a different round key is added to each of the two new words, producing the two output words of the F-function.

The g-function may be implemented in a variety of ways, depending upon one's choice of keying strategy. Twofish defines five different keying strategies: Compiled, Full, Partial, Minimum, and Zero. These choices enable a wide range of time/memory trade-offs for a Twofish implementation.

For RISC and EPIC microprocessors, the choice of Full keying is the most natural. Full keying requires 4096+128+32 = 4256 bytes of table for the four key-dependent S-boxes, 32 round keys, and eight whitening keys. This table size poses no problem for a modern computer platform. Compiled keying is able to reduce the Twofish Pentium-Pro encryption time from 315 cycles to 258 cycles, but it necessitates a separate copy of the encryption and decryption codes for each different key. For superscalar RISC and EPIC microprocessors, Compiled keying is unlikely to result in a performance gain. Given sufficiently many general registers, key loading always can be overlapped and executed in parallel.

The heart of the Twofish g-function is defined as:

1. Partition the 32-bit input word into four 8-bit bytes.
2. Use the value of each of the four bytes to index and fetch a new byte value from a corresponding, 256 byte, key-dependent S-box.
3. Matrix multiply the MDS matrix, a predefined, maximal distance separation byte matrix by the vector of the four bytes fetched from the S-boxes. Scalar multiply of bytes in GF(2^8) is represented as GF(2)[x] modulo v(x), where v(x) is the primitive polynomial $x^8+x^6+x^5+x^3+1$. Scalar addition of bytes in GF(2^8) is XOR.

For Full keying, each of the four S-boxes contains 256 32-bit words, rather than 256 8-bit bytes. Each 32-bit word of S-box$_{32}$[i] is the four-byte vector computed by matrix multiplication of the MDS matrix by the four-byte vector whose sole non-zero component is the byte S-box$_8$[i]. If we denote matrix multiplication by +.×, and the bytes of a column vector, least significant byte first, as [B0:B3] or [B0, B1, B2, B3] the 32-bit S-boxes are:

$$\begin{aligned}
\text{S-box0}_{32}[\text{i}] &= \text{MDS } +.\times\ [\text{S-box0}_{8}[\text{i}], 0, 0, 0] \\
\text{S-box1}_{32}[\text{i}] &= \text{MDS } +.\times\ [0, \text{S-box1}_{8}[\text{i}], 0, 0] \\
\text{S-box2}_{32}[\text{i}] &= \text{MDS } +.\times\ [0, 0, \text{S-box2}_{8}[\text{i}], 0] \\
\text{S-box3}_{32}[\text{i}] &= \text{MDS } +.\times\ [0, 0, 0, \text{S-box3}_{8}[\text{i}]]
\end{aligned}$$

In this manner, all GF(2^8) byte multiplications of the g-function MDS matrix multiply are pre-computed, and saved in the 32-bit S-boxes. With these S-boxes, all that is required for a g-function MDS matrix multiplication is to fetch a 32-bit word from each of the four S-boxes and XOR the words together. Therefore, the Full keying computation of the g-function consists of extracting four 8-bit bytes from the input word, using each extracted byte to index and fetch a 32-bit word from a corresponding S-box, and XORing the four fetched words. The rotation by eight bits of the right input word to the F-function actually requires no explicit computation. It is accomplished simply by the order in which 8-bit bytes are extracted from the input word. Similarly, no computation is required for word swapping between rounds.

Keying

Full keying for a Twofish 128-bit user-supplied key proceeds in three phases. In each phase the approach taken utilizes modestly sized tables to accelerate the performance. The user-supplied key is taken as four 32-bit words, in little-endian byte order. These words are called M_0, M_1, M_2, and M_3. Their byte contents, respectively are: [m_0:m_3], [m_4:m_7], [m_8:m_{11}], and [m_{12}:m_{15}], where m_i is the i'th byte of the user-supplied key.

In the first phase of keying, two four-byte vectors denoted S_0 and S_1 are derived from the user-supplied key. These vectors are utilized in the computation of the S-boxes. S_0 and S_1 each are computed by a matrix multiplication of the RS matrix by an eight-byte vector of user-supplied key bytes. The 4×8 RS matrix is derived from a Reed-Solomon code, and is specified by the Twofish definition. Specifically:

$$S_0 = [\text{RS}] +.\times [m_0:m_7] \qquad\qquad S_1= [\text{RS}] +.\times [m_8:m_{15}]$$

For the RS matrix multiplication, scalar multiply of bytes in GF(2^8) is represented as GF(2)[x] modulo w(x), where w(x) is the primitive polynomial $x^8+x^6+x^3+x^2+1$. Scalar addition of bytes in GF(2^8) is XOR. The actual computation of these two matrix multiplications is accomplished by simulating the LFSRs for the RS code. Doug Whiting programmed this in the following manner in the original Twofish submission.

[6] Schneier, Kelsey, Whiting, Wagner, Hall, Ferguson, *The Twofish Encryption Algorithm*, John Wiley & Sons, 1999.

```
#define      RS_GF_FDBK   0x14D        /* field generator */
#define      RS_rem(x)                                                \
{ BYTE  b  = x >> 24;                                                 \
  DWORD g2 = ((b << 1) ^ ((b & 0x80) ? RS_GF_FDBK : 0 )) & 0xFF;      \
  DWORD g3 = ((b >> 1) & 0x7F) ^ ((b & 1) ? RS_GF_FDBK>>1 : 0) ^ g2;  \
         x = (x << 8) ^ (g3 << 24) ^ (g2 << 16) ^ (g3 << 8) ^ b;
}
```

S_0 and S_1 then can be calculated by the following triply-nested loop, where `M[i]` denotes M_i and `S[i]` denotes S_i:

```
for( i = 0; i < 2; ++i ) {
    for( j = 0, r=0; j < 2; ++j ) {
        r ^= (j) ? M[i*2] : M[i*2+1];
        for( k = 0; k < 4; ++k ) {
            RS_rem( r );
        }
        S[i] = r;
    }
}
```

The calculation of S_0 and S_1 can be accelerated by using a pre-computing a table of 32-bit words, `RStbl[256]`, where `RStbl[i]` = `RS_prem(i)`. `RS_prem(x)` is identical to `RS_rem(x)` but without the `(x << 8)` term in the final assignment statement. Each cycle of the LFSRs then may be simulated simply by:

```
unsigned int x;
#define   RS_rem(x)  x = (x << 8) ^ RStbl[x >> 24];
```

The triply-nested loop to compute S_0 and S_1 is completely unrolled. Housekeeping instructions may be executed in parallel with this computation.

The second phase of keying is to compute the four key-dependent S-boxes. Four pre-computed, 256 entry, 32-bit word auxiliary tables are utilized to accelerate this computation. These tables, denoted `MD0`, `MD1`, `MD2`, and `MD3`, are similar to the Full key S-boxes. Two additional 256 entry, 8-bit tables are required for the S-box computation. These are the tables containing the basic q0 and q1 byte permutations defined in the Twofish specification. These tables are denoted `q0` and `q1`. Each auxiliary table entry combines the final q0 or q1 byte permutation of the S-box computation, and the MDS matrix multiplication. Specifically :

```
MD0[i]  =  MDS +.× [q1[i], 0, 0, 0]
MD1[i]  =  MDS +.× [0, q0[i], 0, 0]
MD2[i]  =  MDS +.× [0, 0, q1[i], 0]
MD3[i]  =  MDS +.× [0, 0, 0, q0[i]]
```

This is the same matrix multiplication used in the g-function. Each Full key S-box contains exactly the same 32-bit words as the corresponding auxiliary table, but permuted according to the user-supplied key. If we designate the bytes of the words S_0 and S_1 as S0(3:0) and S1(3:0), byte zero being least significant, the Full key S-box computation loop is:

```
for( i = 0; i < 256; ++i ) {
   S-box0_32[i] = MD0[ q0[ q0[i]^S0(0) ] ^ S1(0) ];
   S-box1_32[1] = MD1[ q0[ q1[i]^S0(1) ] ^ S1(1) ];
   S-box2_32[i] = MD2[ q1[ q0[i]^S0(2) ] ^ S1(2) ];
   S-box3_32[i] = MD3[ q1[ q1[i]^S0(3) ] ^ S1(3) ];
}
```

This computation further can be accelerated by yet another, 256-entry, auxiliary 32-bit word table. This table is called `q0q1q0q1`. The i'th entry of this table consists of `[ q0[i], q1[i], q0[i], q1[i] ]`. The word `q0q1q0q1[i]` can be fetched by a single instruction, and can be XORed with S_0. This computes the inner XOR of all four assignment statements in parallel. Each byte of this intermediate result then is used to fetch a byte from q0 or q1. Following one more XOR with the corresponding byte of $S_{1,}$ the S-box$_{32}$ entry is obtained by indexing and fetching the 32-bit word from the proper MD table. This word is stored into the proper 32-bit S-box.

The code to perform this computation is organized as a 256-pass loop for both PA-RISC and IA-64. The S_0 and S_1 words already reside in general registers. For each loop iteration, the required operations are one indexed load for the q0q1q0q1 table entry, a word XOR with S_0, four byte extracts[7], four indexed byte loads from the q0 and q1 tables, four XORs with S_1 bytes, four indexed word loads from the MD tables, four indexed word stores to the S-boxes, and a loop closing instruction. For IA-64, eight additional instructions are required for computing table addresses. IA-64 post address modification is used for indexing the q0q1q0q1 table and the S-boxes.

The total number of 256-entry tables used to accelerate the computation of S_0, S_1, and the key-dependent S-boxes is eight, occupying 6656 bytes. These table sizes are quite acceptable for a modern RISC or EPIC platform. No IA-64 bank optimization was done for these tables[8]. No additional tables are required for the third phase of keying.

1.	q0	256 bytes
2.	q1	256 bytes
3.	q0q1q0q1	1024 bytes
4.	MD0, …, MD3	1024 bytes each, 4096 bytes total
5.	RStbl	1024 bytes

[7] The four S_1 byte extracts are done outside the loop.

[8] Described in the next section.

The third and final phase of keying is the computation of the 40 whitening and round keys. This code is similar to the computation of the S-boxes. It is organized as a 20-iteration loop, in which two keys are computed per iteration. Unlike the S-box computation, each key requires a full MDS matrix multiply. Further, a final PHT transform is applied to each pair of keys. The Twofish definition systematically uses the same MDS matrix multiply and PHT operations in the keying algorithms and in the encryption and decryption algorithms.

The same table techniques used above are used to accelerate computation of the whitening and round keys. The initial eight of the 40 keys are taken as the input and output whitening keys. The final 32 keys are taken as the round keys. Using the previously defined notations, and `K` to denote the newly computed keys, the computation for the 40 whitening and round keys is:

```
for( i = 0; i < 40; i += 2 ) {
   T0  = MD0[ q0[ q0[i]^M2(0) ] ^ M0(0) ];
   T0 ^= MD1[ q0[ q1[i]^M2(1) ] ^ M0(1) ];
   T0 ^= MD2[ q1[ q0[i]^M2(2) ] ^ M0(2) ];
   T0 ^= MD3[ q1[ q1[i]^M2(3) ] ^ M0(3) ];
   T1  = MD0[ q0[ q0[i+1]^M3(0) ] ^ M1(0) ];
   T1 ^= MD1[ q0[ q1[i+1]^M3(1) ] ^ M1(1) ];
   T1 ^= MD2[ q1[ q0[i+1]^M3(2) ] ^ M1(2) ];
   T1 ^= MD3[ q1[ q1[i+1]^M3(3) ] ^ M1(3) ];
   T1  = (T1 <<< 8);
   T0 += T1;
   T1 += T0;
   T1  = (T1 <<< 9);
   K[i] = T0;
   K[i+1] = T1;
}
```

The code to perform this computation is organized as a 20-pass loop for both PA-RISC and IA-64. Note that the M_i words are used in even-subscript and odd-subscript pairs. Also note that the M_i words are used in an order reversed from the order of the S_i words in the S-box computation. The $M_{0,}$ M_1, M_2, and M_3 words already reside in general registers. For each loop iteration, the required operations are two indexed loads for the q0q1q0q1 table entries, two word XORs with M_2 and M_3, eight byte extracts[9], eight indexed byte loads from the q0 and q1 tables, eight XORs with M_0 and M_2 bytes, eight indexed word loads from the MD tables, six XORs to complete the MDS matrix multiplies, two rotates, one add and one shift-and-add for the PHT, two indexed word stores to the key array, and a loop closing instruction. For IA-64, sixteen additional instructions are required for computing table addresses. IA-64 post address modification is used for indexing the q0q1q0q1 table and the key array.

Encryption

For PA-RISC the encryption and decryption functions are organized as straight-line code. Each is provided two pointer arguments, the first to the 16-byte cleartext block or ciphertext block, the second to the concatenation of the round keys, whitening keys, and four Full key S-boxes. Input blocks are whitened 64-bits at a time. Housekeeping instructions are overlapped with the first and last rounds.

Each PA-RISC round, including the one-bit circular shifts, executes in about a dozen cycles. PA-RISC includes an instruction that can extract any contiguous 8-bit field from a word in one cycle. The extracted byte can be used directly as an index for a 32-bit word load instruction. Further, the PA-RISC shift-and-add instruction permits the PHT to be done in two instructions during the same cycle. Thus, each round needs 32 instructions: eight extract instructions (`extrw,u`), eight instructions to load from S-boxes (`ldw,s`), two instructions to load round keys (`ldw`), two one-bit circular shift instructions (`shrpw`), eight XOR instructions (`xor`), three add instructions (`add,l`), and one shift-and-add instruction (`shladd,l`). The instruction schedule is nearly optimal, but the final right rotate by one bit adds one cycle to the round.

For IA-64 the encryption and decryption functions are organized in exactly the same way. In each round, an additional instruction is required to compute an S-box address from each extracted byte. Although this requires eight additional instructions, there also is an added benefit. Microprocessor caches often are organized as independent 8-byte banks. An optimal memory strategy, therefore, shuffles the four S-boxes, so that each S-box is entirely contained in a single cache bank. This results in a 16-byte stride between successive S-box words. The IA-64 shift-and-add instructions, used to compute S-box addresses, therefore, use a shift value of four. This assures the absence of cache bank conflicts when executing two S-box loads during the same cycle.

A second technique employed for IA-64 is computational height reduction, a practice common for parallel instruction issue machines. Additional instructions are executed, but the entire computation completes in fewer cycles.

In Twofish, the rightmost bit of the first F-function output becomes the high order bit of a byte to be extracted in the next round. For PA-RISC, the fact that the extract instruction demands a contiguous bit field requires that the one-bit right rotate be done after computation of the first F-function output and prior to the extract for the next round. For IA-64, parallelism and predicates offer a better solution.

The first F-function output is computed as three XORs, two adds, and a final XOR. Although these operations do not commute or freely associate, they in fact do so for the rightmost bit, which actually is the result of six XORs. By computing the rightmost bit of the last XOR sooner (round-key XOR third-block-word), one redundantly can compute the rightmost bit of the first F-function output one cycle earlier. This permits the rightmost bit also to be tested without adding a cycle to the round. The result of the test is written to a predicate. This predicate then is used to set a temporary S-box pointer either to the beginning, or to the halfway point, of the corresponding S-box at the start of the next round. Only the seven leftmost bits of the unrotated first F-function output are extracted in

[9] The eight M_0 and M_1 byte extracts are done outside the loop.

the next round. They then are used as an index relative to the temporary pointer. The full first F-function output word can be rotated later.

It also turns out that, with proper table alignment, height reduction can be used to compute two S-box addresses one cycle earlier in the next round. The enabling fact here is that offsets into S-boxes consist of 12 bits, of which the right four are zero. For a 4096-byte aligned and shuffled table, an XOR can be used for the address calculation. The terms for two such XORs redundantly can be computed in the previous round. This can be seen from the following equations for one pair of encryption terms (note: [7:0] denotes the rightmost 8 bits of a word):

Let: PHT_1be the second PHT output for the current Round.

RK_1 be the second Key word for the current Round.

BW_3 be the fourth Block word for the current Round.

Fin_1 be the second input word to the next Round.

PSB1 be the pointer to S-box 1.

pSBE be the pointer to the S-box 1 entry for Fin1[7:0].[10]

$$Fin_1 = (PHT_1+RK_1) \oplus BW_3$$

$$\begin{aligned} pSBE &= pSB1 + 16*(Fin_1)[7:0] \\ &= pSB1 + 16*((PHT_1+RK_1) \oplus BW_3)[7:0] \\ &= pSB1 + (16*(PHT_1+RK_1)[7:0] \oplus 16*BW_3[7:0]) \\ &= pSB1 \oplus (16*(PHT_1+RK_1)[7:0] \oplus 16*BW_3[7:0]) \quad [11] \\ &= (pSB1 \oplus 16*BW_3[7:0]) \oplus (16*(PHT_1+RK_1)[7:0]) \end{aligned}$$

Performance

Cycles	**Pentium**	**PA-RISC**			**IA-64**	**IA-64++**
		Min	Average	Max		
Keying	8414	2846	2901.79	2964	2445	2445
Encryption	315	205	217.45	233	182	182
Decryption	311	200	210.29	224	182	182

On PA-RISC, Twofish keying executes in 2846 cycles, compared to the best-reported Pentium results of 8414, a 2.96:1 performance advantage. Encryption and decryption also run faster: a 36% advantage for encryption (205 vs. 315) and a 35.7% advantage for decryption (200 vs. 311).

On IA-64, Twofish executes even faster. Twofish keying executes in 2445 cycles, compared to the best-reported Pentium results of 8414, a 3.44:1 performance advantage. Encryption and decryption also run faster: a 42.2% advantage for encryption (182 vs. 315) and a 41.5% advantage for decryption (182 vs. 311).

[10] S-box 1 is used for the rightmost bits because of the logical (Fin_1 <<< 8).

[11] Addition is equivalent to exclusive-or because of the S-box table alignment.

Conclusions

All the algorithms have reasonable implementations on PA-RISC and IA-64; all make good use of the architectures. It is clear that the underlying computer architecture has a direct and significant effect on the optimal implementation for each candidate. The large register files in PA-RISC and IA-64 enable complete state to be kept without using memory, influencing the structure of Rijndael, Twofish, and keying codes. The choice of equations for Serpent is a direct result of the available execution width and ALU operations. Sometimes, effects are expressible only at the assembly level, such as the software pipelines in the Mars keying or the MMU multiplication in RC6 encryption. In other cases, algorithm structures to exploit the underlying architecture are best expressed in high level source, such as the restructuring of the RC6 keying algorithm.

Our second conclusion is that *algorithm performance cannot be measured by a single number*. A complete performance characterization must filter out large system effects such as caching, memory latencies, interrupts, paging, process swaps, and I/O activity, but should draw attention to fine-grain system effects such as cache interference and execution latencies. When timing keying for random input key values, the results will exhibit a performance distribution rather than a single number.

Another consideration is parallelism. Future CPU's will be increasingly, and we believe explicitly, parallel; algorithms that can exploit parallelism will see continuing performance improvement over the life of the new AES algorithm. It should be observed that as better Serpent equations are developed, Serpent will further improve both its performance and parallelism. A final factor in evaluating software is memory usage; none of the finalists use tables uncomfortably large for modern server and desktop systems.

Using these criteria, and assuming that the IA-64++ additions will/will-not be made, the results of this study rank the AES finalists as follows:

Performance	Memory	Parallelism
Rijndael	RC6	Rijndael
RC6/Twofish	Serpent	Twofish
Twofish/RC6	Mars	Serpent
Mars	Twofish	Mars
Serpent	Rijndael	RC6

Acknowledgments

We wish to express our thanks to Doug Whiting for his unerring guidance, especially his prescient counsel in the selection of first-round candidates to investigate. We also wish to thank and to acknowledge the contributions of Dr. Brian Gladman, whose work is well known and appreciated by the AES community. Brian's codes were used to generate test values and, in many instances, improved our understanding of the algorithms. Brian also kept us up to date on his Pentium performance improvements. We appreciate Rohit Bhatia's suggestions for 32×32 multiplies. Dag Arne Osvik contributed his Serpent equations, which forced our equation analysis tools to improve and sped up both the PA-RISC and the IA-64 Serpent implementations.

We are indebted to John Crawford and members of the Intel Itanium team, who provided access and support for the Itanium simulator. Finally, our thanks go to the Hewlett Packard Ft. Collins McKinley team, whose assistance with the development and simulation tools, and patience with our endless questions, was the *sine qua non* of the IA-64 work.

Appendix A:
Summary of Best Performance

Candidate	Encryption					Decryption					Keying				
	Clocks	Ops	IPC	Regs	Bytes	Clocks	Ops	IPC	Regs	Bytes	Clocks	Ops	IPC	Regs	Bytes
Mars															
Pentium	320					374					3894				
New Keying											2128				
PA-RISC	540	631	1.17	12(18)	2588	538	632	1.17	12(18)	2592	1969	2908	1.48	20	2584
New Keying	538	631	1.17	12(18)	2588	537	632	1.17	12(18)	2592	1797	1805	1.00	20	1984
IA-64	511	1013	1.98	18//8	784	527	1013	1.92	18//8	784	1903	3332	1.75	14//48	1344
New Keying											1408	3132	2.22	12//16	976
IA-64++	255					271					1313				
New Keying											1408				
Table Sizes					2048					2208					2208
Alg Parallelism			2.0					2.0					3.0		
RC6															
Pentium	243					226					1632				
PA-RISC	580	577	0.99	12(4)	2308	493	558	1.13	12(4)	2232	1077	1519	1.41	12	760
IA-64	490	826	1.69	4/27/8	480	490	826	1.69	4/27/8	528	1581	2629	1.66	8//56	256
IA-64++	150					130					1057				
Table Sizes					0					176					176
Alg Parallelism			2.0					2.0					2.0		
Rijndael															
Pentium	284					283					1338				
PA-RISC	168	537	3.20	24	2160	168	539	3.21	24	2160	239	686	2.87	28	2800
Fwd Keying											85	228	2.68	19	1504
IA-64	125	704	5.63	20/12	3808	126	706	5.60	20/12	3824	148	822	5.55	24/21	4480
Fwd Keying											104	282	2.71	19	1504
IA-64++	same					same					same				
Table Sizes					8192					8368					8368
Alg Parallelism			10.0					10.0					10.0		
Serpent															
Pentium	900					885					1301				
PA-RISC	580	1273	2.19	17	5100	585	1309	2.24	17	5240	668	1409	2.11	19	5640
IA-64	565	1517	2.61	24	8480	631	1546	2.45	24	8848	475	1527	3.21	22/4	8368
IA-64++	468					407					380				
Table Sizes					0					528					528
Alg Parallelism			3.0					3.0					4.0		
Twofish															
Pentium	315					311					8414				
PA-RISC	205	548	2.67	20	2192	200	548	2.74	20	2192	2846	8904	3.13	30	1324
IA-64	182	927	5.09	23	5184	182	915	5.03	23	4960	2445	9561	3.91	26/21	1600
IA-64++	same					same					same				
Table Sizes					6656					4256					4256
Alg Parallelism			6.0					6.0					4.0		

Notes:

-- IA64++ is a *hypothetical* IA-64 implementation – refer to the text for details. It does not represent any current or planned IA-64 implementation.

-- Twofish times for Full keying are from: *The Twofish Encryption Algorithm*, John Wiley & Sons, 1999.

-- Pentium, Alpha clocks are lowest reported clocks from the NIST Round 1 Report, August 1999.

-- Regs = GRs, or statics/stacked, or statics//rotating, or statics/stacked/rotating, or GRs(FRs) registers.

-- Bytes are object code sizes. Table Sizes are total tables for *keying*, key table plus look-up tables for *encryption* and *decryption.*

-- Alg Parallelism is an estimated integral upper bound for software parallelism.

Appendix B: Mars Keying
Original Implementation

The original Mars keying initializes the first seven elements of an array, `T[-7..39]`, to the first seven entries of the Mars S-box, then sets the rest of the array as follows:

```
T[i] = ((T[i-7] ⊕ T[i-2]) <<< 3) ⊕ k[i mod N] ⊕ i    i = 1 ... 38
T[39] = N
```

where k is the input key and `N` is the size, in words, of the input key. This recurrence has an active state of seven words, `A`, `B`, …, `G`, such that the expansion can be rewritten:

```
T[0]  = A = ((A ⊕ F) <<< 3) ⊕ k[0] ⊕ 0
T[1]  = B = ((B ⊕ G) <<< 3) ⊕ k[1] ⊕ 1
T[2]  = C = ((C ⊕ A) <<< 3) ⊕ k[2] ⊕ 2
T[3]  = D = ((D ⊕ B) <<< 3) ⊕ k[3] ⊕ 3
T[4]  = E = ((E ⊕ C) <<< 3) ⊕ k[0] ⊕ 4
                 . . .
T[38] = D = ((D ⊕ B) <<< 3) ⊕ k[2] ⊕ 38
T[39] = N
```

where `A` is initialized to `S[0]`, `B` to `S[1]`, and so forth. When key expansion is complete, the data words are then "stirred" seven times as follows:

```
T[i] = (T[i] + S[T[i-1] & 0x1ff]]) <<< 9       i = 1 ... 39
T[0] = (T[0] + S[T[39] & 0x1ff]]) <<< 9
```

It is possible to overlap the first stirring with the key expansion: after the first eight expansion steps, `T[1]` is no longer involved in the expansion recurrence and can therefore be stirred. This requires adding one extra word to the expansion state so that both `T[i]` and `T[i-1]` are available for stirring. After the stirring, the keys are reordered, mapping `T[i]` → `K[7i mod 40]`

PA-RISC

The PA-RISC implementation uses straight-line coding for the expansion/stir phase, rotating the key words each step. The remaining six stirring passes are executed in a loop as per the specification. The reordering, however, is again straight-line code. If reordering is considered as replacement rather than a permuted copy, the replacements form chains, that is:

```
T[1] → T[7] → T[9] → T[23] → T[1]
```

There are 8 chains of four, 3 chains of two, and two chains of one (`T[0]`→`T[0]` and `T[20]`→`T[20]`). Since PA-RISC can issue two memory operations per cycle but can retire only one store per cycle, the optimal ordering loads from one chain, then interleaves the stores with the loads from the next chain. The expected performance is 40 cycles, which is the number of times a multiple cycle loop would have to run to perform the same task. It also eliminates the need for a temporary key array: the target key array can be used for all intermediate values.

IA-64

The IA-64 Mars keying implementation takes advantage of the large register files, rotating registers, and rotating predicates. The routine allocates a 48-register stack frame, all of which are rotating. The initial register usage is as follows:

r32-r34	r35	r36	r37	r38	r39	r40	r41	r42	r43	r44	r45	r46	r47	r48	r49	r50-r79
Unused	k_X	k_3	k_2	k_1	k_0	T_i	T_{i-1}	T_{i-2}	T_{i-3}	T_{i-4}	T_{i-5}	T_{i-6}	T_{i-7}	A	B	T[10..39]

The first nine computations simply initialize `T`$_i$ and rotate registers to the right. After that, the registers `A` and `B` contain the first two values for stirring. Unlike PA-RISC, this phase of the computation is enabled by the rotating predicates, where a '1' is shifted in each time through the main body of the loop. To circulate the key words, $k_0 \rightarrow k_X$ at the end of the loop. When the initialization phase of the loop is finished, the loop switches to the epilogue phase, which now shifts a '0' into the rotating predicates, which disables the initialization instructions. Thus, the entire expansion/mix phase executes in one loop that runs 48 times, 6 cycles per loop.

When the first phase is finished, the intermediate key values are in the rotating registers, with `r39` = `T[0]`, `r38` = `T[1]`, …, `r32` = `T[7]`, `r79` = `T[8]`, …, `r48` = `T[39]`. This allows the stirring phases to compute on the rotating register file. Since the registers rotate 39 places during the stirring loop, the registers used in each phase are:

Pass	**`T[i]`**	**`T[i-1]`**	**`T[0](Final)`**
2	r39	r38	r78
3	r78	r77	r69
4	r69	r68	r60
5	r60	r59	r51
6	r51	r50	r42
7	r42	r41	r33

The reorder is efficiently handled in a two cycle loop. In the first cycle, the key word is stored, the data pointer incremented seven words, and a look-ahead target index counter is tested for overflow and incremented. In the second cycle, the index and data pointers are adjusted if the index had overflowed in the previous cycle.

A comparison of AES candidates on the Alpha 21264

Richard Weiss
VSSAD Labs
Compaq Computer Corp,
334 South St
Shrewsbury, MA 01545
Richard.Weiss@Compaq.com

Nathan Binkert
Computer Science Dept
University of Michigan
Ann Arbor, MI
binkertn@umich.edu

ABSTRACT

We compare the five candidates for the Advanced Encryption Standard based on their performance on the Alpha 21264, a 64-bit superscalar processor. There are several new features of the 21264 that have a significant impact on encryption/decryption speed. The main ones are greater potential for instruction-level parallelism (ILP) and larger level 1 cache. The ILP comes from the fact that the 21264 can issue four integer instructions per cycle. We envision that for high-performance servers, there will be multiple streams of data for encryption or decryption. The type of parallelism that we consider in this paper is the encryption of multiple, independent blocks interleaved in the same code loop running on the *same processor*. This benefits some algorithms more than others. Rijndael and Twofish turn out to be the fastest for a single block at a time, but RC6 is potentially the fastest when processing two blocks at a time. The reason for this is that out-of-order execution together with an issue width of four can be used to hide the latency of integer multiplies.

Introduction

The new AES algorithms will be used on a wide range of CPU's. The Alpha 21264 is a good representative of a 64-bit RISC architecture. Its features include a 64K two-way set associative level-1 cache, the capability to issue 4 integer instructions each cycle, and out-of-order execution. Since the Alpha is most likely to be used in servers, it will probably be used for encrypting or decrypting multiple streams of data simultaneously. This can be done on multiple processors, but it is also relevant to look at the efficiency of processing more than one block simultaneously on each processor, thus increasing the throughput of the system. In the remainder of this paper, we will use the term ***multiple stream*** or ***multistream*** to refer to more than one block on the same processor. Most of the studies so far have looked at single stream performance, where latency is the dominant factor. In order to get optimal multistream performance, it will be necessary to harness the full bandwidth of the processor. The five candidate AES algorithms have different computational requirements, and therefore have different behavior with respect to multistream than single stream.

We illustrate the multiple stream scenario with an example, so that there is no ambiguity. Consider the following assembly language fragment from a loop for an imaginary processor that can issue two instructions per cycle, at most one of which can be a multiply:

```
loop:
      1. Load S[0]         # load key
      2. T = Mull A*A

      3. Load S[1]         # load key
      4. U = Mull B*B

      5. C = Shift_right  T
      6. D = Shift_left   T

      7. E = Shift_right  U
      8. F = Shift_left   U

      9. C = C Or D
      10. E = E Or F

      11. B = C Add S[0]
      12. A = E Add S[1]

      13. Br loop
```

The processor will execute two instructions per cycle except for the branch. If the latency of each instruction were one cycle, then the whole code would take seven cycles. However, if the latency of a multiply is seven cycles and at most one can be issued in a given cycle, then there is a five cycle stall after the fourth instruction. Therefore, the execution time increases to 12. Now consider what we can do for two independent blocks of data:

```
loop:
      Load S1[0]         # load key1
      T1 = Mull A1*A1

      Load S1[1]         # load key1
      U1 = Mull B1*B1

                        C2 = Shift_right  T2
                        D2 = Shift_left   T2

                        E2 = Shift_right  U2
                        F2 = Shift_left   U2

                        C2 = C2 Or D2
                        E2 = E2 Or F2

                        B2 = C2 Add S2[0]
                        A2 = E2 Add S2[1]

                        Load S2[0]         # load key2
                        T2 = Mull A2*A2

                        Load S2[1]         # load key2
                        U2 = Mull B2*B2

      C1 = Shift_right  T1
      D1 = Shift_left  T1

      E1 = Shift_right  U1
      F1 = Shift_left   U1

      C1 = C1 Or D1
      E1 = E1 Or F1

      B1 = C1 Add S1[0]
      A1 = E1 Add S1[1]

      Br loop
```

The combined loop can process two blocks in only 13 cycles. The processing of the two blocks can be overlapped in such a way that while the shift operations for one block are waiting for the multiplies to complete, operations on the other block can proceed. For the 21264, the latency for a multiply is actually seven, and the latency of a load is three or more, depending on whether or not the value is in the D-cache. The 21264 can issue up to four integer instructions in one cycle, at most two of which can be loads. The out-of-order processing capability is not actually used if the compiler schedules the instructions to take into account the latency. It should be noted that future generations of Alpha processors will have simultaneous multithreading (SMT), which will eliminate the necessity of the programmer/compiler merging two streams of data in one instruction stream.

The key to taking advantage of the full issue width of the Alpha is recognizing when a program has a low number of instructions per cycle (ipc). In the above example, this was caused by the long latency of the multiplies, but there may be other cases where this happens. For example, in the implementation of Serpent that we used, there were long chains of dependent logical operations, which resulted in an ipc of slightly less than two. Thus, Serpent can achieve a speedup of almost two by processing two streams. RC6 is similar to the example above in that the multiplies introduce latency, which reduces the ipc to a level for which processing two streams works well. On the other hand, Rijndael, Twofish and Mars do not lend themselves to this approach. They can be coded efficiently for single stream so that the table lookups can be overlapped with the other computation and the ipc is well over two. It should be noted that an ipc of greater than two does not preclude multistream processing, but the gains are likely to be small. Also, it is important to use an optimized version of the code, otherwise a low ipc will only reflect the inefficiency of the implementation rather than the potential for multistream parallelism. For this reason, we examine assembly language implementations in addition to the C versions.

One of the architectural features that is missing from Alpha is the 32-bit rotate. This requires several instructions to emulate. A fixed rotation requires two shifts an “and” and an “or”. These can be executed in two parallel chains and in the absence of other parallelism they have an ipc of two.

The next section presents an analysis of each algorithm in terms of ipc for a C implementation and for an assembly code implementation.

Analysis of Algorithms

Our goal is to get a quick estimate of the performance for multistream data. We do this by checking the timings for the Gladman C implementations of the five candidate algorithms for single stream data and estimating the ipc. Then in some cases, we also look at assembly language implementations to see if the ipc could be increased. While a high ipc will rule out a gain from multistream, a low ipc does not guarantee one. A range of techniques was used from a complete implementation in assembly language in the case of Rijndael, to coding a single round in assembly language for Rc6 and Twofish, to a data dependency anlysis for Mars and Serpent. The data dependency analysis together with instruction latency was used to estimate optimal times for the last two algorithms. In the one case where we did an assembly language implementation, the time for this was compared with our estimate. Finally, we estimated the gains for multiple stream implementations.

Mars

The Mars algorithm has three phases: simple arithmetic and logical operations, table lookup and rotations. The table lookup, which is mixed with some fixed rotations has a four-fold parallelism. This seems to be the reason for a high ipc, and therefore little gain from multistream. Since the Alpha does not have a 32-bit rotate, this increases the number of instructions. For this reason, it is both one of the fastest algorithms on a Pentium Pro but one of the slowest on the 21264.

RC6

RC6 turns out to be a lot more efficient on the Alpha 21264 than expected from observing the number of cycles for a single block of data. For single stream performance, each round when coded in assembly language, takes 18 cycles and there are 20 rounds. If we allow 20 cycles for setup, this gives a total of 380 cycles per block. This is amazingly close to the current reported figure of 382 cycles per block for the optimized C version. A single round of encryption for two independent blocks of data simultaneously was also coded in assembly language for an estimated 21 cycles, which is less than 11 cycles/block. For 20 rounds, this would be 210 cycles/block plus the time for setup and storing results. This is as fast as Rijndael, and is potentially more consistent since it uses multiplication, which have a fixed latency, and does not depend on table lookups which could suffer occasional cache misses. In addition, if the algorithm were used with a word size of 64, this could potentially double the throughput, since the 64-bit versions of the operations multiply, xor, add and rotate are as fast or faster than the 32-bit versions on Alpha processors.

Rijndael

The simplicity of the Rijndael algorithm makes it easy to analyze. We were able to produce an efficient implementation in assembly code together with timing results. The major computational cost for this algorithm is accessing the look-up tables. This can be done in three instructions: extract byte, add to base address, and load the value. For Alpha, this is relatively fast, since the tables fit in the level-one cache. Ideally, one round of Rijndael could be done in 18 cycles: however, in practice, this requires tuning the code to eliminate I-cache misses, D-cache misses, etc. What we observed was that the code took 246 cycles/block when executed repeatedly. This is about 23 cycles per round. This was the fastest algorithm we have observed for 128-bit key length. However, since the number of rounds for Rijndael depends on the key length, this is not the fastest for all applications.

We expect the Rijndael algorithm to scale well with future processors since the makeup of the code is such that one quarter of the instructions are loads. The Alpha 21264 can issue four integer instructions per cycle, and there is a four-fold parallelism from the four S-boxes. However, this gives it a high ipc and means that there is little gain from multistreaming. A single round of

Rijndael takes 18 cycles. The setup and exit code adds another 30 cycles to the total to give approximately 210 cycles per block.

Serpent
Based on the C-code from Brian Gladman, this algorithm is the slowest. However, it speeds up very well with multistreaming. The S-boxes are implemented by sequences of bit-parallel logical operations. Due to data dependencies in this code, the ipc is slightly less than two. The technique for estimating the two stream performance was to modify the C code. Each round is composed of three macros: an “xor” with the key, an S-box computation, and a linear transform. The processing of the two streams was interleaved by repeating each macro for the first stream with the identical macro for the second stream. The compiler was able further mix the instructions to eliminate stalls. Nevertheless, Serpent remains one of the slower algorithms because of the large number of rounds and the large number of instructions per round. It should be noted that most of the operations in Serpent operate on bits in parallel. It should be possible to process two blocks of 32-bit words by using the full 64-bit data path. Namely, one block would use the upper 32 bits, and the other block would use the lower bits. There would be an extra “and” for the rotates as well as packing the two words together, but the speedup could be close to 2x.

Twofish
Based on an assembly language coding of a single round, twofish performs approximately as well as Rijndael on both the 21164 and the 21264 for 128-bit key length. Since Twofish does not require more rounds for larger key lengths, its relative performance would be better for longer keys. It can potentially do eight S-box lookups in parallel for each round. This gives it a high ipc and small gain for multistreaming.

Timing Results

Table 1 shows the results from optimized C-code for the Alpha 21164 and 21264 processing one block at a time. The 21164 can issue two integer instructions per cycle and the 21264 can issue four. The results are similar to those published by Granboulan [Gran]. Our timings were all obtained by running each of the algorithms for key setup, encryption and decryption on a single stream of data, one block at a time. The C-versions of these algorithms are the ones published by Gladman [Glad1]. We ported them to Alpha by using the native cycle count register and modifying the declarations to eliminate alignment errors in the code. The basic idea is to time the execution of the encryption (decryption) code running once, then time it running twice. The minimum times over a large number of iterations are subtracted to measure the time to execute the code without the startup costs. In addition, the encryption (decryption) code is run once at the beginning to warm up the caches.

In order to relate our assembly code estimates to the C implementations, we linked our assembly version of Rijndael to the Gladman harness and observed an encryption time of 280 cycles/block. The assembly code when executed for a large number of iterations took a minimum of 246 cycles/block. This suggests that the C++ overhead for calling some of the C or assembly functions could be significant.

In Table 2, we have estimated timing results for assembly language implementations for some of the algorithms for single stream. Table 3 shows the estimated timing for assembly code for processing multiple streams.

EV56 (21164)	Mars	RC6	Rijndael	Serpent	Twofish
Ours	701c	571c	439c	984c	442c
Granboulan website	507c	559c	490c	998c	490c

EV6 (21264)	Mars	RC6	Rijndael	Serpent	Twofish
Ours	515c	428c	293c	854c	316c
Granboulan website	450c	382c	285c	855c	315c

Table 1. Timing comparison in cycles/block for C code.

EV6 (21264)	Mars	RC6	Rijndael	Serpent	Twofish
Assembly code	375c	360c	210c	570c	255c

Table 2. Estimated timing for assembly code in cycles/block.

EV6 (21264)	Mars	RC6	Rijndael	Serpent	Twofish
Assembly code	375c	210c	210c	506c	255c

Table 3. Estimated time for assembly code encrypting two blocks simultaneously. Times are in cycles/block.

Conclusions

RC6 has the most potential for parallelism when multiple streams are processed on the same processor simultaneously in a single thread. One reason for this is that it relies heavily on multiplication, which itself has a large degree of parallelism for the Alpha processors. 32-bit multiplies are inherently parallel because they operate on four bytes at the same time. Using 64-bit multiplication would afford even more parallelism. The 21264 can issue one multiply every cycle. The latency of seven cycles does not limit bandwidth for this algorithm in multistream mode. An S-box lookup requires three instructions, and only operates on one byte at a time. Note that while RC6 has variable 32-bit rotations, one of the intermediate results from the fixed rotation by 5 is re-used in the variable rotation.

Serpent also has a large gain from multistream processing because of the long dependent chains of instructions and low ipc. However, because of the large number of rounds and instructions per round, it still is slow.

Following RC6 are Twofish and Rijndael, which both use 8-bit table lookups and linear transforms. Twofish has an advantage for longer keys, but Rijndael seems the fastest for 128-bit keys. Based on an assembly language implementation of

Rijndael, there can be a significant difference between the estimated performance and what can be readily achieved/observed by counting cycles outside of the algorithm function call. Comparing code execution with timing estimations can have a significant amount of error.

Since our estimates for the Alpha 21264 are based on instruction level parallelism for processing multiple streams, similar behavior should be observable for Itanium and other VLIW machines.

Acknowledgements.
We would like to thank Dr. Brian Gladman for publishing unified C implementations of the five AES candidate algorithms. Also we thank Steve Root for assembly language implementations of some of the algorithms.

References

[KA] Almquist, Kenneth. “AES Candidate performance on the Alpha 21164.
http://home.cyber.ee/helger/aes/kenneth.txt

[Glad1] Gladman, Brian. “Implementation experience with AES candidate algorithms.” Second AES Conference, Feb, 1999.
http://jya.com/bg/gladman.pdf

[Glad2] Gladman, Brian.
http://www.btinternet.com/~brian.gladman/cryptography_technology/Aes/index.htm

[Gran] Granboulan, Louis. “AES Timings of best known implementations.”
http://www.dmi.ens.fr/~granboul/recherche/AES/timings.html

[SKW] Schneier, B., Kelsey, J., Whiting, D., et al. “Performance Comparison of the AES Submissions.”

Performance Evaluation of AES Finalists on the High-End Smart Card

Fumihiko Sano* Masanobu Koike* Shinichi Kawamura† Masue Shiba*

* Toshiba System Integration Technology Center
3-22, Katamachi Fuchu-shi, Tokyo, 183-8512, JAPAN
† Toshiba Research and Development Center
1, Komukai Toshiba-cho, Saiwai-ku, Kawasaki, 210-8582, JAPAN
{fumihiko.sano, masanobu2.koike, shinichi2.kawamura, masue.shiba}
@toshiba.co.jp

Abstract. This paper reports the performance of the AES finalists, MARS, RC6, Rijndael, Serpent, and Twofish, on the high-end smart card that has a Z80 core with Toshiba's arithmetic coprocessor.

1 Introduction

During the first round of AES candidate assessment, some reported the performance evaluation of the algorithms on low-end smart cards. Their reports are important for understanding performance of each AES candidates in memory and computing resource-restricted environments. However, there are, so called high-end smart cards, which are equipped with a specific hardware for accelerating cryptographic processing. In general these cards are less restricted in their resource than low-end smart cards. So, it is important for better understanding of the AES candidates to evaluate the performance on high-end smart cards. NIST as well expressed their interests in such evaluation in [11]. This paper describes our experience in implementing five AES finalists, and summarizes the performances on our high-end smart card available from Toshiba[17].

The high-end smart card is substantially different from low-end one in that its core is equipped with a crypto coprocessor. It may usually correct to say that the amount of memory for a high-end card is larger than that of low-end one. In some cases, however, venders supply cards with large memory amount suitable for their specific purposes regardless of the core. Therefore, we distinguish between high-end and low-end cards based on the type of core.

At first, we present the architectures of the core on our smart card that includes a CPU and a coprocessor architecture. Next, we describe coding rules for our implementation and then, present experiences of five AES finalists accompanied by results of 64-bit ciphers such that DES[10] and MISTY1[9] on our smart card for reference purpose. Finally, we summarize advantages and disadvantages for each implementation.

2 Platform

High-end smart cards available now are usually equipped with 8/16-bit microprocessor and a crypto coprocessor, or accelerator for cryptographic operations[7]. To evaluate the AES finalists' performance on high-end smart cards, we choose Toshiba's T6N55 chip shown in table 1. The chip is equipped with Z80 microprocessor and a coprocessor. The coprocessor is under the control of Z80 and it carries out arithmetic/logical operations when Z80 asks to do so. The coprocessor is originally designed to accelerate the large integer arithmetics. As will described shortly, it is also suitable to accelerate some operations required to implement AES finalists.

Table 1. Features of Toshiba's T6N55 chip

CPU	Z80
ROM	48KB
RAM	1KB
EEPROM	8KB
Max. of Modulus	2,048-bit
Internal Clock Frequency	5MHz

2.1 Z80 Architecture

The Z80 is a famous 8-bit architectured microprocessor developed by ZiLOG[15]. It has an 8-bit accumulator and a flag register, six 8-bit general-purpose registers, two 16-bit index registers, a stack pointer (SP), and a program counter (PC). An accumulator A and a flag register F can be paired and dealt with as if it is a 16-bit register AF. Similarly, 8-bit registers can be paired with particular registers as BC, DE, and HL. Z80 incorporates dual register banks. Each register bank has each register sets such as an accumulator, a flag register, and six 8-bit registers. Note that one can use only one side of the banks at a time. If one want to use registers belonging to the other side of the bank, he should change the contexts with an EXX operation.

The instruction set includes the following classes:

- Load 8-bit values to registers or an accumulator.
- Load 16-bit values to registers.
- Arithmetic or logical instructions for the accumulator with registers.
- A single bit shift or rotate instructions.
- Compare, block transfer, and search instructions.
- Branch instructions.
- Subroutine calls and returns from them.
- I/O instructions.

– Checking or setting a single bit in registers.

There are some particular instructions for extended registers or control instructions of processor. Z80 can execute addition, subtraction, AND, OR, exclusive or (XOR), and single-bit rotation and shift. It does not have instructions for multiplication and division.

On using the ordinary Z80 core, we should take some features of its architecture into account. It needs four clocks even for the basic instructions, such as a no operation (NOP) or a load instructions between registers (LD r, r'). The next fastest instructions, such as for loading a value to a register (LD r, n) consume seven clocks. Operations for 16-bit register sets are more time consuming. Although we try to use faster operations, the average number of clocks needed for an instruction is about six.

2.2 Crypto Coprocessor

The coprocessor is developed mainly to accelerate the processing of the public key cryptosystem. It has 512-byte RAM area (we call it the 'CRAM' area). That area is segregated into two 256-byte RAM areas. The coprocessor can execute various operations between the 256-byte RAM areas or on the 512-byte RAM. Each maximum size of arithmetical operations supported by the crypto coprocessor is shown in table 2.

It can execute the following classes of calculations:

- Addition, subtraction, multiplication, division, and logical operations.
- Modular multiplication.
- Modular exponentiation.
- Montgomery multiplication.
- Extended Euclidean algorithm.
- Memory transfer in CRAM area.

Here, the logical operations mean AND, OR, and exclusive OR(XOR). The memory transfer is used to transfer data on CRAM area efficiently. So, the feature is similar to the direct memory access (DMA). The most time consuming operation is a modular exponentiation with a large exponent. Other operations, when used in implementing AES finalists, are very fast and finish within a time for the minimum execution time of a Z80 instruction.

The coprocessor executes logical operations between operands located on each CRAM areas. Before executing these operations, Z80 have to put several bytes of control words on the CRAM area in addition to the operands. Since Z80 does not perform so fast to the data on memory, using coprocessor operations are efficient for large data, but not so much for small data.

3 Implementations

3.1 Coding Rules

When we implement the AES finalist, we apply the following rules for the coding.

Table 2. Features of Toshiba's Crypto Coprocessor

Instruction	Max. of Operands (bits)
Addition	2,048
Subtraction	2,048
Multiplication	1,024
Division	2,048
Modular Multiplication	1,024
Exponentiation	1,024

– Program codes are located on the ROM area, and we do not change the code at any time.
– We can use all registers, i.e., registers on both sides of the banks.
– The codes run in constant time not depend on the data to avoid timing analysis.
– We can use memory on the CRAM area if necessary.
– We write codes that generate the extension keys with on-the-fly, if possible.

A time constancy of a code is an imprecise term. We try to give more precise idea behind the third rule. If we have only to realize the time-constancy, we may choose an easy way to stretch the execution time by merely adding NOPs at the end of the code. But what we really have to do is to avoid timing analysis. So, we have to pay more attention not to leak meaningful information. If we can successfully apply the third rule, we can prevent simple power analysis as well as timing attack. The third rule is not sufficient, though it seems necessary, to prevent the differential power analysis. We don't discuss on the differential power analysis in this paper any further.

It is interesting that we may neglect the differences between rounds, for example the key expansion of DES need 2-bit rotations in some rounds. They may leak some information, but it seems useless for analysis.

In this section, we report the performance of AES finalists in alphabetic order. For comparison purpose, results for 64-bit block ciphers, such as DES, triple DES, and MISTY1, will be shown, as well. We describe the speed of each algorithm with clocks and RAM requirement: In each table, 'Int.' means that size of required CRAM for coprocessor's operations, and 'Ext.' means other work area. Note that 5,000 clocks at 5MHz correspond to 1 millisecond. For example, DES needs about 25,000 clocks, and thus it works in 5ms.

The code of DES does not necessarily obey the coding rules above since some permutations for DES are realized by hard wired logic. The triple DES is a two-keyed one, but it executes the key schedule three times with on-the-fly. Therefore, three-keyed triple DES will have the same performance result. MISTY means the MISTY1 algorithm[9] with eight rounds.

To apply our results easily for other processors that have similar features, we try to reduce the memory usage on the CRAM area. But, in this paper, we see that the memory usage is of little importance, since the platform chosen has sufficient memory for these implementations.

3.2 MARS

It is the most difficult task for us to implement MARS on smart cards or other limited resources. MARS has a complex high level structure such as eight rounds of unkeyed forward mixing, eight rounds of keyed forward transformation, eight rounds of keyed backward transformation, and eight rounds of unkeyed backward mixing. Each of the eight rounds consists of so called type-3 Feistel network. In a type-3 Feistel network, input data is segregated into four words. One of them is taken as a pseudo-random function's input and the output is used to modify three other data words. Since MARS has a block length of 128 bits, each word has 32 bit length.

There are three disadvantages of MARS when implemented on a smart card. The first is that it needs 2KB table for S-boxes, but it is not serious. The second is the weakness check of extended key on the key schedule. The last is the rotations with variable shift amount. We discuss the last two disadvantages here.

It is necessary for MARS to implement complicated "weak" measures on the key schedule[3]. The weak keys for MARS are different from those of DES. In the case of DES, you may disregard the problem of weak key because it only increases some potential threats caused by the weak key properties. However, in the case of MARS, since the weak key check procedure is a part of the algorithm specification, you have to check the weak on the key schedule certainly. Otherwise, you may see a terrible result, such as differences in cipher text, although it encrypts the same plain text with common key. As mentioned above, the function of checking the weak on the key schedule is primarily needed.

Although implementing weak key check is necessary, it is also true that this introduces another problem for smart card implementation. If we check the weak and regenerate extension keys, there is a risk of applying timing attack. The regeneration of extension keys causes difference in processing time and leaks some information on the key. Further study of coding is necessary to avoid this problem.

To save our time, our implementation just omits the weak key check. Therefore, it is not complete. Our implementation is not so slow because of customization for 256-bit key and omitting to check 'weak' on the key schedule. The codes for check 'weak' on the key schedule will increase the requirement of ROM and processing time.

The rotations depend on a key data or an internal data are crucial for Z80 or other 8-bit processors since we need to write codes that run in constant time, or else an attacker can get some information about the key. Fortunately, our coprocessor can operate modular multiplications over any modulus. We use them for rotations. Modular multiplications on our smart card are very fast, and finish within a single instruction of Z80. It means that we can operate modular multiplications and data dependent rotations in a constant time and avoid timing attack.

It seems that MARS is a prudent algorithm against cryptanalysis. But it causes some difficulties in implementing on smart cards or similar resource-restricted environments.

Table 3. MARS

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	60	36	24	3,977	45,588
Schedule	512	512	0	1,491	21,742
Total	512	512	24	5,468	67,330

3.3 RC6

RC6 has various parameters and is defined as RC6-$w/r/b$ where w means the word length, r means the number of rounds, and b means the length of key with bytes. We write the code with the recommended parameters for AES such as RC6-32/20/32.

RC6 has a simple structure, but the round function includes various operations such as, addition, subtraction, multiplication, and rotations depending on a variable data. Most part of RC6 constructed by arithmetical operation. Therefore, we operate almost all operations on the coprocessor. Furthermore, since the coprocessor can operate up to 1,024 bits for operand, we can execute the pair of rotations with constant shift amount in parallel. An n-bit rotations to two data is written as follows: We duplicate each of data and put them on corresponding CRAM area, then multiply them with 2^n. As a result, we can improve the performance and reduce the size of code.

The coprocessor can execute RC6 data encryption efficiently. RC6 has a simple key schedule but need much iterations and does not suitable with on-the-fly. The key schedule takes four times as long execution time as encryption.

There is an idea to improve the key schedule processing time. A precomputed table improves the speed, but increase the size of code. It omits the computation of 43 initial values (S[i]) with 32-bit word. The modified code copies S[i]s from precomputed ROM table to RAM area instead of computing S[i]s with constant values. It shall reduce about 4,000 clocks. It needs some extra code or table for precomputed table, thus the size of code increases about 150 bytes.

On the smart cards, RC6 has a moderate encryption speed among the finalists, but its key schedule is slower than Rijindael or Twofish. Note that it has been reported that on the 32-bit processor, RC6's performance is faster than Rijndael and Twofish[5].

Table 4. RC6

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	124	124	0	489	34,736
Schedule	90	90	0	571	138,851
Total	156	156	0	1,060	173,587

3.4 Rijndael

256-bit key is the fastest for on-the-fly key generation, since we can translate the internal key every two rounds. 128-bit key is a little slower than 256-bit key, since we need to make extension keys every round. In the case of 192-bit key, since the key length is not the multiple of the block length, it is not so easy to implement on-the-fly key generation.

The `xtime` is an important subroutine for time constancy. It needs modulus operation with the primitive polynomial. Here is an example of straightforward implementation of the `xtime(a)` algorithm where the original value is stored in A register.

```
        RLA
        JR    NC, SKIP
        AND PRI          ; PRI means the primitive polynomial.
SKIP:
        ...              ; end.
```

This is a very dangerous code. Since ‘AND *PRI*’ operation is operated only when the carry is ‘1’, an attacker can know whether the value excesses 2^8 or not in this code. We must avoid such an implementation. Therefore, we use some techniques to avoid differences of processing time and thus prevent cryptanalysis using timing attack. Here is an example of `xtime(a)` operation with constant time, where `a` is stored in A register.

```
RLA
LD      B, A
SBC     A, A
AND     PRI
XOR     B
```

RLA is a instruction of 1-bit leftward rotation for A register. If RLA is carried out, MSB of A register is set to the carry flag. ‘SBC A, A’ is an instruction which substract a value in A register and a carry from A register. It means that if the carry flag is ‘1’ then A register has a value `0xff`, otherwise A register has a value `0x00`. Next we operate AND instruction with *PRI* for A register. Then we get *PRI* or a value `0x00` in A register, and we can operate whether ‘XOR PRI’ or ‘NOP’ with the same instructions and processing time.

The transformation MixColumn is implemented in an efficient way shown in section 5.1 in [4]. We implement the `AddRoundKey` and data transfers with the coprocessor. Other transformations in Rijndael are not so heavy even for only the Z80 core. Rijndael is the most efficient algorithm on the finalists on our smart card.

A disadvantage of Rijndael is that it needs another code for decryption because of the asymmetry of encryption and decryption. If you need both encryption and decryption algorithms, it takes twice ROM area for code since most part of it cannot be shared.

Table 5. Rijndael

	RAM (bytes)			ROM (bytes)	Time (clocks)
	Total	Int	Ext		
Encrypt	34	32	2	700	25,494
Schedule	32	32	0	280	10,318
Total	66	64	2	980	35,812

3.5 Serpent

There is two kinds of implementation of Serpent: ordinary implementation and bitsliced implementation. Here is the result of an ordinary implementation of Serpent. It is not a bitsliced implementation. It needs a 2,048-byte ROM table on the ordinary implementation.

Serpent has various rotational operations. As is described in MARS implementation, modular multiplication with coprocessor can be used if they improve the performance. Most of the rotations are, however, more efficient with the Z80 operations than with the coprocessor. 1-bit leftward or rightward rotations can be implemented with the Z80 operations, and shifts with multiplies of 8-bit are reorder of bytes. We use the coprocessor operations only for 11-bit rotations, XOR, and memory transfer. Due to the architecture of our coprocessor, it is not suitable to efficiently implement three-operand operation used in Serpent.

In [2], Serpent can be implemented using under 80 bytes of RAM with on-the-fly. Our implementation needs twice more RAM, because we write it with coprocessor's operation XOR between halves of CRAM with different offsets.

It has more rounds than other finalists do, so its performance is not so good as Rijndael or Twofish.

The bitsliced implementation will reduce the size of code and required RAM with a little degradation in speed. In memory-restricted environment, bitsliced implementation may be better than the ordinary coding. In this paper, we attach importance to the speed. So, we choose the ordinary implementation for performance comparison.

3.6 Twofish

In the case that the length of key is less than 256-bit, we need to pad out the original key until it becomes 256-bit. We implement Twofish with 128-bit key to

Table 6. Serpent

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	68	68	0	3,524	71,924
Schedule	96	96	0	413	147,972
Total	164	164	0	3,937	219,896

take the processing time for padding into account. It includes code for padding, and it is a little slower than 256-bit key.

There are two models for implementing Twofish, such as Feistel model and non Feistel model[14]. We implement it with non Feistel model. We assume that it is faster than Feistel. We use coprocessor's operations for additions with subkeys, XOR, and memory transfers on CRAM area, but rotations are implemented with Z80's rotations.

The performance of Twofish depends on the size of precomputed tables' [14]. We consider that the case of using some tables amounted to 1,536 bytes. This code is compact for processing the key schedule with precomputed tables. It seems be compatible with 2200 bytes for code and table size model in [14]. The size of precomputed tables is belongs to encryption code in table 7.

Twofish is as fast as DES on throughput. It does not have any exceptional advantages, but we have nothing to complain about the performance.

Table 7. Twofish

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	34	32	2	2,493	31,877
Schedule	56	32	24	315	28,512
Total	90	64	26	2,808	60,389

4 Summary

We summarize the performance and the required resources on our implementations in table 8. The RAM includes required byte in the RAM area and the CRAM area. Note that when using a coprocessor, the required amount of RAM increase, because of the alignment rules for CRAM area.

Some finalists are designed to have heavy key schedules. They are intended to prevent exhaustive search attacks, but resulting in speed reduction on smart cards. We consider that Rijndael is excellent on all aspects. RC6 is as good as Rijndael on the code point of view, but the key schedule consumes more time.

Twofish needs much ROM memory than RC6 and Rijndael because of the table. It is faster than Triple DES and equal to DES on the throughput. It will have good performance on any smart cards. MARS has disadvantages of its code size caused by four of eight round iterations and a 2,048-byte table. The speed is equal to Twofish's one. We consider MARS has some difficulties to check 'weak' on the key schedule and regenerate. Serpent has disadvantages of its performance caused by the iterations of rounds and the difficulty of key schedule. The bitsliced implementation will improve the requirement of ROM or RAM, but slower than others.

We tried to write all program codes to consume as little RAM area as possible. On the other hand, if we may regard the RAM area, especially CRAM area, as a kind of free work space, it will be unfair to compare finalists how little work area they consume. Nevertheless, notice that MARS consumes all the CRAM area, whereas others consume at most half of the area.

Table 8. Comparison of AES finalists and the algorithms

Cipher	RAM		ROM		Time (clock)						
	(bytes)		(bytes)		Encrypt		Schedule		Encrypt + Schedule		
MARS	572	5	5,468		45,588	4	21,742	2	67,330	3	*
RC6	156	3	1,060	2	34,736	3	138,851	4	173,587	4	
Rijndael	66	1	980	1	25,494	1	10,318	1	35,812	1	only encryption
Serpent	164	4	3,937	4	71,924	5	147,972	5	219,896	5	
Twofish	90	2	2,808	3	31,877	2	28,512	3	60,389	2	
DES	17		772						25,398		
Triple DES	17		849						72,341		
MISTY	44		1,598						25,486		

$*$: omit to check "weak" in the key schedule.

5 Conclusion

We have implemented AES finalists on a high-end smart card that is equipped with a crypto-coprocessor. The resulting code has higher performance than that on a low-end smart cards, since multiplication and rotation are efficiently implemented using the coprocessor's commands. Coprocessor's RAM are also useful for work memory, as well.

Regarding speed, Rijndael is the best one and is as fast as our DES implementation. It is twice faster than DES on the throughput. RC6 is suitable for our smart card same as on the 8051[6, 8], but not to be compared with Rijndael or Twofish because of the key schedule.

For smart card implementation, it is necessary to perform key schedule at least for every processing block, in order to save memory areas to store extended key. For the same reason, it is desirable for key schedule to be suitable for on-the-fly key generation. As a result, design concept for key schedule affects the performance very much, and those algorithms that have heavy key schedule are not advantageous for smart card implementation.

Finally, we report the performance of E2[12] that is a candidate on the first round in the appendix.

References

1. R. Anderson, E. Biham, and L. Knudsen, *"Serpent: A Proposal for the Advanced Encryption Standard"*, AES submission, 1998.
2. R. Anderson, E. Biham, and L. Knudsen, *"Serpent and Smartcards"*, CARDIS '98, 1999, available on http://www.cl.cam.ac.uk/~rja14/serpent.html.
3. C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr., L. O'Connor, M. Peyravian, *"MARS -a candidate cipher for AES"*, AES submission, 1998.
4. J.Daemen, V.Rijmen, *"AES Proposal: Rijndael"*, AES submission, 1998.
5. B. Gladman, "AES Algorithm Efficiency",
http://www.btinternet.com/~brian.gladman/cryptography_technology/Aes/
6. G. Hachez, F. Koeune, and J. Quisquater, *"cAESar results: Implementation of Four AES Finalists on Two Smart Cards"*, The second AES conference, 1999, available on http://www.dice.ucl.ac.be/crypto/CAESAR/caesar.html.
7. H. Handschuh, and P. Paillier, *"Smart Card Crypto-Coprocessors for Public-Key Cryptography"*, CryptoBytes, Vol. 4, No. 1, RSA Laboratories, 1998.
8. G. Keating, *"Performance Analysis of AES candidates on the 6805 CPU core"*, The second AES conference, 1999,
available on http://www.ozemail.com.au/~geoffk/aes-6805/.
9. M. Matsui, *"New Block Encryption Algorithm MISTY"*, Fast Software Encryption, 4th International Workshop Proceeding, LNCS **1267**, Springer-Verlag, 1997, pp.54-68.
10. National Bureau of Standards, *"Data Encryption Standard"*, U.S.Department of Commerce, FIPS 46-3, October 1999.
11. J. Nechvatal, E. Barker, D. Dodson, M. Dworkin, J. Foti, and E. Roback, *"Status Report on the First Round of the Development of the Advanced Encryption Standard"*, http://csrc.nist. gov/encryption/aes/round1/r1report.pdf
12. Nippon Telegraph and Telephone Corporation, *"Specification of E2 – a 128-bit Block Cipher"*, AES submission, 1998.
13. R.L. Rivest, M.J.B. Robshaw, R. Sidney, Y.L. Yin, *"The RC6 Block Cipher"*, AES submission, 1998.
14. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, *"Twofish; A 128-Bit Block Cipher"*, AES submission, 1998.
15. ZiLOG, *"Z80 Microprocessor Products"*,
available on http://www.zilog.com/products/z80.html
16. http://csrc.nist.gov/encryption/aes/round2/Round2WhitePaper.htm, 1999.
17. http://www.toshiba.co.jp/about/press/1999_02/pr_j0301.htm, (in Japanese).

A E2

E2 is not selected as a finalist for the second round review. But it has a good performance, especially encryption speed without key schedule. The serious disadvantages of E2 are that it has time consuming key schedule and can't execute it with on-the-fly. Fortunately, since the RAM usage fits on the half of CRAM area, we select a way to extend all round keys on the half of them, at first. In this case, E2 is efficient for encryption just like the report in [6]. The round function is designed as suitable for byte oriented operations. It is good for the Z80 architecture. It is, however, difficult for Z80 to execute multiplication on the IT and division on the FT. We use the coprocessor's commands for these operations. Those commands include XOR, memory transfer, multiplication, and inverse.

Table 9. E2

	RAM (byte)			ROM (byte)	clock
	Total	Int	Ext		
enc	26	24	2	1,519	17,018
key	548	512	36	296	79,358
Total	548	512	36	1,815	96,376

How Well Are High-End DSPs Suited for the AES Algorithms? [*]

AES Algorithms on the TMS320C6x DSP

Thomas J. Wollinger[1], Min Wang[2], Jorge Guajardo[1], Christof Paar[1]

[1]ECE Department
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609, USA
Email: {wolling, guajardo, christof} ece.wpi.edu

[2] Texas Instrument Inc.
12203 S.W. Freeway, MS 722
Stafford, TX 77477, USA
Email: minwang micro.ti.com

Abstract

The National Institute of Standards and Technology (NIST) has announced that one of the design criteria for the Advanced Encryption Standard (AES) algorithm was the ability to efficiently implement it in hardware and software. Digital Signal Processors (DSPs) are a highly attractive option for software implementations of the AES finalists since they perform certain arithmetic operations at high speeds, they are often smaller and more energy-efficient than general purpose processors, and they are commonly used for the rapidly growing market of embedded applications. In this contribution we investigate how well modern high-end DSPs are suited for the five final candidates chosen after the second AES conference. As a result of our work we will compare the optimized implementations of the algorithms on a state-of-the-art DSP.

Keywords: cryptography, DSP, block cipher, implementation

[*] This research was supported in part through a graduate fellowship by secunet Security Networks AG and a grant from the Texas Instrument University Research Program.

1 Introduction

The National Institute of Standards and Technology (NIST) has initiated a process to develop a Federal Information Processing Standard (FIPS) for the Advanced Encryption Standard (AES), specifying an encryption algorithm to replace the Data Encryption Standard (DES) which expired in 1998 [14]. NIST has solicited candidate algorithms for inclusion in AES, resulting in fifteen official candidate algorithms of which five have been selected as finalists. Unlike DES, which was designed specifically for hardware implementation, one of the design criteria for the AES candidate algorithms is that they can be efficiently implemented in both hardware and software. Thus, NIST has announced that both hardware and software performance measurements will be included in their efficiency testing. Several earlier DSP's contributions looked into the software implementation of the AES algorithms on various platforms [1]. However, there was only one publication dealing with the implementation of the candidate algorithms on a Digital Signal Processor (DSP) [9].

Digital Signal Processors are a distinct family of micro processors. In comparison to the more common general purpose processors such as those offered by, e.g., Intel and Motorola, DSPs allow for fast arithmetic, special instructions for signal processing applications, real-time capabilities, relatively lower power, and relatively lower price (obviously, those statements tend to over-generalize and should not be taken too literally). The main application areas of DSPs are embedded systems, such as wireless devices, cable and Digital Subscribe Line (DSL) modems, various consumer electronic devices, etc. With the predicted increase of embedded applications and pervasive computing, it is not unreasonable to expect that DSPs and DSP-like processors will become more commonplace. At the same time, it seems likely that many future embedded applications will need some form of encryption capability, for instance, for assuring privacy over wireless channels.

The questions that we try to address in this contribution are: How well are high-end DSPs suited for the implementation of the AES finalists? Can modern DSPs compete with general purpose computers in terms of speed?

In this paper, we focus on the implementation of the five AES finalists on a Texas Instruments TMS320C6000 DSP platform. In particular, the implementations are on a 200 MHz 'C62x/'C64x which performs up to 1600/8800 million instructions per second (MIPS) and provides thirty-two/sixty-four 32-bit registers and eight independent functional units.

2 Pre ious ork Cryptography on SPs

The field of implementing cryptographic algorithms on special platforms is very active. However, the research done on implementation of cryptographic schemes on a DSP is limited. There are a few papers that deal with public-key cryptography. There is one previous paper about the implementation of the AES candidates on a DSP. The papers [3, 7, 10] deal primarily with the implementation of public key algorithms on DSP processors. The main conclusion of these papers is that DSPs are a good choice for these algorithms due to the integer arithmetic capabilities of DSPs.

Reference [7] also describes the implementation of DES on a Motorola DSP 56000. It was found that the algorithm encrypts at roughly the same speed as a contemporary PC (20 MHz Intel 80386).

Karol Gorski [9] commented on the set of the AES Round 1 candidate algorithms, based on the timings obtained on the TI TMS320C541 DSP. Reference [9] used the C implementation by Brian Gladman, compiled with full compiler level optimizations. The resulting low speeds of the algorithms were due to the 'C54x 16 bit operations which are not ideal for most of the AES candidates. There was also no effort made to optimize the algorithms beyond those optimizations automatically performed by the C compiler.

3 Methodology

3.1 The Implementation of the Five AES Finalists

We implemented Mars, RC6, Rijndael, Serpent and Twofish on a TMS320C6201 DSP. RC6 was also implemented on the C64x DSP. As the basis of the implementations we used either the reference or optimized C code provided by the algorithm's authors, or the C code written by Brian Gladman [8].

It is important to point out the way we chose to code each algorithm, because they all offer several implementation options. In [6], the authors of Rijndael proposed a way of combining the different steps of the round transformation into a single set of table lookups. Each table has 256 4-byte word entries. Similarly, our Twofish implementation uses the "Full Keying" option as described in the specification [13]. Inother words we used 4 KByte tables which combine both the S-box lookups and the multiplication by the column of the MDS matrix. RC6 is a fully parameterized encryption algorithm [11]. The version of RC6 that we implemented is RC6-32/20/16. Mars was coded in the original version as stated in the algorithm specifications in [4], with 8, 16, and 8 rounds of "forward mixing", "main keyed transformation", and "backwards mixing", respectively. Finally, in [2] the authors described an efficient way to implement Serpent. Thus, we implemented the S-boxes as a sequence of logical operations which were applied to the four 32-bit input blocks.

3.2 Tools and Optimization Effort

The source code was first compiled using the standard Texas Instruments C compiler (versions 3.0 and 4.0 alpha), utilizing the highest level of optimizations (level 3) available. For further information about the levels of optimization performed by the compiling tools, see [15, page 3-2 and 3-3].

After the implementation of the C code version, we optimized the encryption and decryption functions of the algorithms so that the compiler could further optimize it. In order to do so, we took advantage of the 32-bit data bus which is capable of loading 32-bit words at a time. We performed math operations with *Intrinsic Functions* to speed up the C code. *Intrinsic Functions* are similar to an additional mathematical Run-Time Support (RTS) library. They allow the C code to access hardware capabilities of the 'C6x devices while still following ANSI C coding practices. We also tried to use as many of the functional units in parallel as possible, e.g., by replacing constant multiplication by shifts, by unrolling loops, or by preserving loops.

We further rewrote the encryption and decryption function for most algorithms in linear assembly to achieve performance improvements. Linear assembly is assembly code that has not been register-allocated and is unscheduled. The assembly optimizer assigns registers and uses loop optimization to turn linear assembly into highly parallel assembly. However, we did not program in pure assembly which is a very challenging and time consuming task on a complex processor such as the 'C6201, with eight independent functional units.

3 3 Parallel Processing Single- lock Mode s Multi- lock Mode

In addition to the optimizations described above, we implemented a second version of code in which data blocks can be processed in parallel. With parallel processing, the encryption and the decryption functions can operate on more than one block at a time using the same key. This allows better utilization of the DSP's functional units which leads to better performance.

With parallel processing, however, the speedups may only be exploited in modes of operations which do not require feedback of the encrypted data, such as Electronic Code-Book (ECB) or Counter Mode. When operating in feedback modes such as Ciphertext Feedback mode, the ciphertext of one block must be available before the next block can be encrypted. For the remainder of our discussion, single-block mode will denote feedback modes and multi-block mode will denote non-feedback modes.

3 4 The TMS32 C62x igital Signal Processor

We chose the TMS320C6201 fixed point digital signal processor out of the TMS320C62x family. In this subsection we introduce the key architectural features of the DSP which are relevant for our implementation.

The 'C6201 performs up to 1600 million instructions per second (MIPS) at a clock rate of 200 MHz. These processors have thirty-two 32-bit registers and eight independent functional units. As shown in Figure 1, the 'C62x has four pairs of functional units. The architecture of the DSP has effectively been divided in two identical halves. Each half is composed of four independent functional units (S, , , and) and a bank of sixteen 32-bit registers. The processor also allows limited communication between the two halves.

The multiplier unit is indicated by and accepts two 16-bit words as an input and outputs a 32-bit result. In addition to the two multipliers, the processor provides six arithmetic logic units (ALUs). The unit, that has the ability to perform 32/40-bit arithmetic operations, comparisons, normalization count for 32/40-bits, and 32-bit logical operations. With the unit we can add 32-bit words, subtract, do linear and circular address calculation, and write to and load from memory. The S unit provides the functionality for 32-bit arithmetic operations, 32/40-bit shifts, 32-bit bit-field operations, 32-bit logical operations, branching, constant generation, and register transfers to/from the control register file [16].

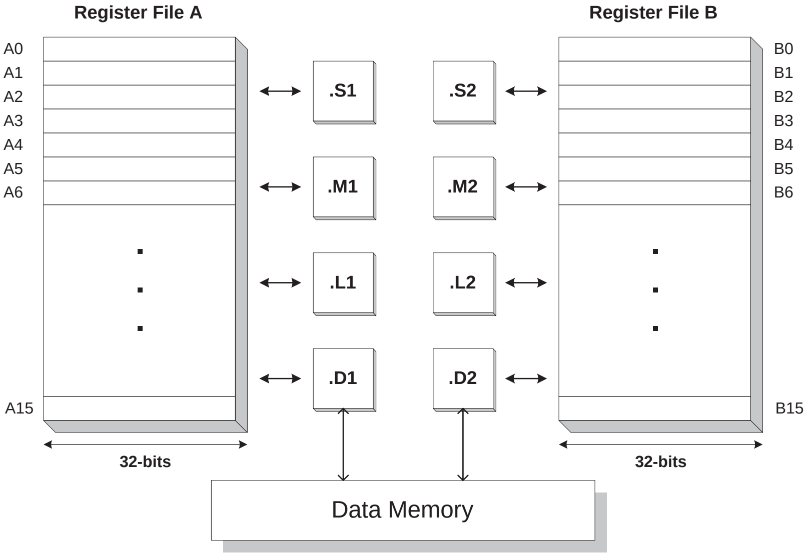


Figure 1: TMS32062x Functional Units [16]

The 'C6201 includes a bank of on-chip memory and a set of peripherals. Program memory consists of a 64K-byte block that is configurable as cache or memory-mapped program space. A 64K-byte block of RAM is used for data memory. The peripheral set includes two serial ports, two timers, a host port interface, and an external memory interface.

The 'C6000 development environment includes: a C Compiler, an Assembly Optimizer to simplify programming and scheduling, and the Code Composer Studio^{TM}, which is a MS Windows debugger interface for visibility into source execution. All of the 'C6000 devices are based on the same CPU core featuring elociTI^{TM}, a highly parallel architecture that provides software-based flexibility and good code performance for multi-channel and multi-function applications.

4 Results

4 1 Results on the TMS32 C62 1 SP

All the figures presented in this section refer to a 128-bit block encryption or decryption with a key of 128 bits. The algorithms are timed with the Code Composer Simulator, which is part of the Code Composer Studio^{TM} for the TMS320C6201 DSP. Code Composer Simulator uses the simulated on-chip analysis of a DSP to gather profiling data.

The reported results in Table 1 refer to either a C or a Linear Assembly implementation. In the cases where we had the possibility to choose between two implementations we referenced the fastest results found by us. All the timings shown are obtained from a C implementation using the compiler version 4.0 alpha unless otherwise indicated.

To convert cycle counts into encryption or decryption rates expressed in bits per second, we divided $128 * 200 * 10^6$ by the cycle count. For example, the encryption speed of Twofish in multi-block mode is computed as: $128 * 200 * 10^6/184 = 139.1$ Mbit/sec.

The order of the algorithms is based on the mean speed of encryption and decryption in multi-block mode. The mean speed can simply be calculated by adding the speed of the encryption and decryption functions and then dividing the sum by two. For instance, the mean speed in multi-block mode for RC6 equals $(128.0 + 116.4)/2 = 122.2$ Mbit/sec.

		DSP		DSP		Pentium-Pro	
		multi-block mode		single-block mode			DSP multi-block
		200MHz		200MHz		200MHz	mode/Pentium
		cycles	Mbit/sec	cycles	Mbit/sec	Mbit/sec	
Twofish	encryption	184	139.1	308	83.1	95.0 [17]	1.5
	decryption	172	148.8	290	88.3	95.0 [17]	1.6
RC6	encryption	200 †	128.0	292	87.7	97.8 [12]	1.3
	decryption	220 †	116.4	281	91.1	112.8 [8]	1.03
Rijndael	encryption	228 ‡	112.3	228 ‡	112.3	70.5 [8]	1.6
	decryption	269 ‡	95.2	269 ‡	95.2	70.5 [8]	1.4
Mars	encryption	285	89.8	406	63.1	69.4 [8]	1.3
	decryption	280	91.4	400	64.0	68.1 [8]	1.3
Serpent	encryption	772	33.2	871 *	29.4	26.8 [8]	1.2
	decryption	917*	27.9	917 *	27.9	28.2 [8]	1.0

Table 1: Performance results of the AES candidates on the TMS320C6201

Here are comments about the results in Table 1:

- The highest level of optimizations were used for all algorithms, with the exception of Serpent decryption. The loop in Serpent is too complex and too long so the optimizer was only able to schedule the code in a lower level. Hence, the performance figures for decryption are slightly worse than the numbers for encryption. In addition, the throughput of the decryption function is the same for single-block and multi-block modes.

- The linear assembly code of Rijndael can be optimized by the tools very efficiently. In this case we could not gain a performance advantage by parallel processing, which results in the same speed for single-block and multi-block modes.

- In all cases, except for RC6 encryption, we encrypted and decrypted two blocks at a time in multi-block mode. We were able to process three blocks at a time in parallel for RC6

*C implementation using compiler version 3.0

†Linear assembly implementation using compiler version 3.0

‡Linear assembly implementation using compiler version 4.0 alpha

encryption. Hence, we could use a large number of functional units in parallel and could reach a high throughput. For some of the other algorithms we tried to use three blocks in parallel as well. However, the optimizer was not able to create efficient loops due to the number of instructions.

4.1.1 Results in Multi-Block Mode

In Table 1 we compare the throughput speeds of the TMS320C6201 and a 200MHz Pentium Pro. In order to allow for an easy comparison we added the rightmost column to the table, where we divided the highest speed in multi-block mode on the DSP with the performance numbers on the Pentium. In this way we normalized our numbers by the speed achieved on the Pentium Pro platform. If the ratio is larger than one, the implementation of the algorithm on the DSP is faster than the one on the Pentium. One can see that in all cases but one we could achieve higher throughput on the DSP than the best known results on a Pentium Pro II with the same clock rate. Only for Serpent decryption were the Pentium and the DSP speeds almost identical.

We can also see from the performance ratio in the rightmost column how well the algorithm structure is suited for the DSP. Rijndael encryption and Twofish decryption gain the most when implemented on the DSP compared to the implementation on a Pentium. In both cases the quotient of the throughputs is approximately 1.5, which means that the speed of the particular function on the DSP is roughly 50 faster than the same function on the Pentium.

In addition to our above analysis, we ranked the AES finalists based on their performance on the 'C6000 DSP family. This ranking compares the mean speed of the algorithms in multi-block mode. Twofish with a mean speed of 144.0 Mbit/sec and RC6 with 122.2 Mbit/sec are the fastest algorithms. These two algorithms are followed by Rijndael with a mean throughput of 103.8 Mbit/sec and Mars with 90.3 Mbit/sec. Serpent with 30.6 Mbit/sec is poor in terms of throughput on the DSP.

4.1.2 Results in Single-Block Mode

The results stated above refer only to the cases in which we used multi-block mode. If we look at the single-block mode case, Rijndael encryption and decryption as well as Serpent encryption perform better on the DSP than on a Pentium. Rijndael encryption with 112.3 Mbit/sec is almost 60 faster than the corresponding Pentium implementation and Rijndael decryption at 95.2 Mbit/sec is almost 40 faster. Judged by their speed performance on the C62x, Serpent decryption, Mars encryption and decryption, and Twofish decryption are slightly worse than on a general-purpose computer. The remaining functions, Twofish encryption and RC6 encryption and decryption, are much slower than the corresponding Pentium functions.

If we had ranked the algorithms based on their mean speed in single-block mode, Rijndael with 103.8 Mbit/sec would be the fastest, followed by RC6 with 89.4 Mbit/sec, and Twofish with 85.7 Mbit/sec. Mars with 63.6 Mbit/sec and Serpent with 28.7 Mbit/sec are not as good in single-block mode.

We would like to point out that all of our “best” results were achieved using the methodology described above, and that other coding styles, such as pure assembly, might be able to achieve higher throughputs.

4.1.3 Comparison of the Results with the Critical Path of the Algorithms

Craig S.K. Clapp analyzes the critical path of Crypton, E2, and the five AES finalists. In his analysis, [5] only counts instructions and cycles associated with the transformation of a plaintext block into a ciphertext block in ECB mode. In other words, instructions associated with loading of plaintext, storing of ciphertext, and loop overhead are ignored. Clapp concludes that based on the length of its critical path, Rijndael stands well ahead of the pack with 71 cycles/block. Twofish (162 cycles/block), RC6 (encryption with 181 cycles/block and decryption with 161 cycles/block), and Mars (214 cycles/block) form the second tier. Finally, Serpent's critical path is a factor of two longer than the next nearest candidate (encryption with ≤ 526 cycles/block and decryption with ≤ 436 cycles/block).

The results that we achieved in single-block mode are in agreement with those obtained by analyzing the critical path. Rijndael is in both cases by far the fastest algorithm. The throughput of RC6 is slightly better than the throughput of Twofish on the DSP, even though the critical path of Twofish is a little shorter than the one from RC6. Mars is ranked in both, the DSP speed analysis and the critical path analysis of [5], the same. Serpent results trail the nearest candidate in both analyses by more than a factor of two. It is important to point out that while the critical path for decryption is shorter than that for encryption in Serpent, decryption is actually slower than encryption in the DSP implementation.

The discrepancies are due to our use of automatic optimization. The optimizer tries to create the best machine code possible. Nevertheless, the optimizer might not be able to reach the cycle count of the critical path for some algorithms. We might be able to overcome these differences by rewriting the functions in full assembly. We were not able to do this because of time constraints.

4.1.4 Memory Usage

Embedded system applications have often memory constrains. Hence this subsection looks at the memory requirements of our implementation. The 'C6201 has three 16 Mbyte regions of external memory. These regions can support synchronous or asynchronous 32-bit access. There is also one 4 Mbyte region of asynchronous external memory which is typically used to store the boot information. The 'C6201 contains one megabit of internal RAM which is split between program and data memory. All this internal memory is zero wait-state. Table 2 summarizes the memory usage of the algorithms in our implementation.

As it can be seen from Table 2, the memory usage of the algorithms varies almost by an order of magnitude. RC6 uses the least program memory and Serpent the most. In some cases, e.g. for Serpent, the algorithms require a large amount of program memory, because we optimized them for speed. Hence we calculated the look-up tables on the “fly” with boolean-algebra and this increases

	Memory Usage multi-block mode		Memory Usage single-block mode	
	Data ROM /Bytes	Program /Bytes	Data ROM /Bytes	Program /Bytes
Mars	3072		3072	
encryption		3280		2428
decryption		2956		2372
RC6	0		0	
encryption		608		576
decryption		672		576
Rijndael	16384		16384	
encryption		2360		1180
decryption		2960		1480
Serpent	0		0	
encryption		5844		3568
decryption		6016		5104
Twofish	168		168	
encryption		1416		700
decryption		1420		708

Table 2: Memory Usage on the TMS320C6201

the program code. The data ROM represents constant arrays, which in our cases correspond to the look-up tables. RC6, for example, uses no tables, hence the data ROM is zero.

4 2 Results on the TMS32 C64x

The TMS320C64x clock can be scaled to up to 1.1 GHz and can perform up to 8800 MIPS. The C64x has extended parallelism support with quad 8-bit and dual 16-bit operations. Also, the sixty-four 32-bit registers and 8 functional units lead to better performance. We also took advantage in our implementation of the better data access and the extended instruction set of the C64x (for example, rotation, Galois field multiplication, etc.).

We chose RC6 to be implemented on the C64x. The results that we present in this section are based on a C implementation and are compiled with compiler version 4.0 beta.

The results in Table 3 for RC6 achieved with the 'C64x in multi- and single-block mode are better than the results we got from the 'C6201. RC6 encryption in multi-block mode is almost 70 faster than on a general-purpose machine.

At this point it is important to remark that the optimizer tools are quite advanced for the 'C62x, but are still in a very early stage for the 'C64x. That means if we only perform C code optimizations, we will not get good performance numbers on the 'C64x. We expect an improvement when we rewrite the functions in linear assembly. We did a detailed analysis for hand coded assembly RC6 and we estimated a performance of 229 cycles/block (for each encryption- and decryption-function) in single-block mode.

		DSP		DSP		Pentium-Pro	
		multi-block mode		single-block mode			DSP multi-block mode/Pentium
		200MHz		200MHz		200MHz	
		cycles	Mbit/sec	cycles	Mbit/sec	Mbit/sec	
RC6	encryption	155	165.2	277	92.4	97.8 [12]	1.7
	decryption	154	166.2	278	92.1	112.8 [8]	1.5

Table 3: Performance results of two AES candidates on the TMS320C64x

5 Conclusions

"How well are high-end DSPs suited for the AES algorithms?" was the main question that we asked ourselves as a motivation to write this paper. We noticed that in almost all cases the AES finalists' encryption and decryption functions reach higher speeds on the 'C6000 DSPs than the best known Pentium Pro II implementations, at identical clock rates. It was observed that some of our implementations on the 'C6201 were over 50 faster than the best known performance numbers on the Pentium platform. In addition, our implementation of RC6 on the 'C64x reached speeds which were almost 70 faster than those of the Pentium. RC6 on the 'C64x encrypts with a throughput of 165.2 Mbit/sec and decrypts with a speed of 166.2 Mbit/sec. Twofish with an encryption speed of 139.1 Mbit/sec and decryption of 148.8 Mbit/sec was by far the fastest throughput that we obtained on the 'C6201. Hence, we can conclude from our results, that state-of-the-art DSPs are well suited for the architecture of the AES finalists.

6 Acknowledgment

We would like to thank William Cammack from TI for his helpful comments.

References

[1] Second Advanced Encryption Standard (AES) Conference. Rome, Italy, March 1999. National Institute of Standards and Technology (NIST).

[2] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[3] P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Processor. In A. M. Odlyzko, editor, *Ad ances in Cryptology Crypto* , volume 263, pages 311 326, Berlin, Germany, August 1986. Springer- erlag.

[4] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko unic. Mars - a candidate cipher for AES. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[5] Craig S.K. Clapp. Instruction-level Parallelism in AES Candidates. Second AES Conference, March 1999. `http://csrc.nist.gov/encryption/aes/reound1/conf2/papers/clapp.pdf`

[6] J. Daemen and . Rijmen. AES Proposal: Rijndael. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[7] Stephen R. Dusse and Burton S. Kaliski Jr. A Cryptographic Library for the Motorola DSP56000. In Ivan B. Damgard, editor, *uroCrypt* , volume 473 of *ecture otes in Computer Science*, pages 230 244, Berlin, Germany, May 1990. Springer- erlag.

[8] Brian Gladman. AES Algorithm Efficiency, 2000.
`http://www.btinternet.com/~brian.gladman/cryptography_technology/Aes2/index.htm`

[9] Karol Gorski and Michal Skalski. Comments on the AES Candidates. Technical report, National Institute of Standards and Technology, ENIGMA SOI Sp. z o.o., Warsaw, Poland, April 1999. `http://csrc.nist.gov/encryption/aes/round1/comments/R1comments.pdf`

[10] Kouichi Itoh, Masahiko Takenaka, Naoya Torii, Syouji Temma, and asashi Kurihara. Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201. In Cetin K. Koc and Christof Paar, editors, *Cryptographic ardware and mbedded Systems*, volume 1717 of *ecture otes in Computer Science*, pages 61 72, Berlin, Germany, August 1999. Springer- erlag.

[11] R. Rivest, M.J.B. Robshaw, R. Sidney, and .L. in. The RC6^{TM} Block Cipher. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[12] RSA Security. The RC6 Block Cipher - Performance, 1999.
`http://www.rsasecurity.com/rsalabs/aes/rc6_performance.html`

[13] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall. Twofish: A 128-Bit Block Cipher. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[14] W. Stallings. *Cryptography and etwork Security.* Prentice Hall, Upper Saddle River, New Jersey 07458, 2nd edition, 1999.

[15] Texas Instruments Incorporated. *S C ptimi ing C Compiler ser s uide.* Custom Printing Company, Owensville, Missouri, February 1998.

[16] Texas Instruments Incorporated. *S C C rogrammer s uide.* Custom Printing Company, Owensville, Missouri, February 1998.

[17] D. Whiting. Twofish Timing Measurements. electronic mail personal correspondence, January 2000.

Fast Implementations of AES Candidates

Kazumaro Aoki[1] and Helger Lipmaa[2]

[1] NTT Laboratories
1-1 Hikarinooka, Yokosuka-shi, Kanagawa-ken, 239-0847 Japan
maro@isl.ntt.co.jp
[2] Küberneetika AS
Akadeemia tee 21, 12618 Tallinn, Estonia
helger@cyber.ee

Abstract. Of the five AES finalists four—MARS, RC6, Rijndael, Twofish—have not only (expected) good security but also exceptional performance on the PC platforms, especially on those featuring the Pentium Pro, the NIST AES analysis platform. In the current paper we present new performance numbers of the mentioned four ciphers resulting from our carefully optimized assembly-language implementations on the Pentium II, the successor of the Pentium Pro. All our implementations follow well-defined API and timing conventions and sensible guidelines, like no using of self-modifying code and key-specific static data — i.e., tricks that speed up the implementation but at the same time restrict the field of application. Our implementations are up to 26% percent faster than previous implementations. Our work also shows how a simple change (inclusion of the MMX technology) in the analysis platform can influence the relative encryption speed of different ciphers. To enable everyone to compare their implementations to ours, we also fully specify our procedures used to obtain the speed numbers.

1 Introduction

For more than 20 years, DES [FIP77] has been a widely employed cryptographic standard. While the best cryptanalytic attacks against DES (differential and linear cryptanalysis) are still highly impractical, during the last years DES has became obsolete for its too short key and block sizes, not withstanding the current advances in computing technology. Motivated by this, NIST initiated a new effort to replace DES as a standard. 21 algorithms were submitted and 15 algorithms were accepted as AES (*Advanced Encryption Standard*) candidates, of which 5 candidates—MARS [BCD$^+$98], RC6 [RRSY98], Rijndael [DR98], Serpent [ABK98], Twofish [SKW$^+$99b]—were chosen to the second round.

However, the AES process was started not only due to the theoretical reasons: there are a few well-known constructions, including 3DES, that seem to have very good security margins. Unfortunately, 3DES, based on the hardware-oriented DES, is unsatisfyingly slow on the modern 32- and 64-bit computer architectures: modern block ciphers are up to 10 times faster than 3DES. Regardless of these ciphers having unproven (even by time) security properties, they are widely used in the industry by pragmatic reasons: hardware applications like 1 GBits/s Ethernet or on-the-fly encryption of 160 MByte/s

SCSI hard disks are requesting for faster ciphers. Clearly, the situation of having a (moderately) secure and (moderately) fast *de jure* standard DES, a (probably) secure and (clearly) slow *de facto* standard 3DES and some fast but with unknown security margin *de facto* standards is not acceptable: there should be a single standard that is both secure and fast. This is one of the reasons why, when inviting the public to propose candidates for the AES, NIST explicitly stated that the new standard should be both "more secure and faster" than 3DES.

While security of the candidates cannot be exactly quantified by the currently known methods, it seems to be easier to measure their speed. However, there is still a lot of ambiguity in answering the question what AES candidate is the fastest. Several papers (including [Lip99,SKW$^+$99a]) have compared AES candidates speed, but since the implementations quoted in them are often incomparable (or based on pure estimations), one cannot make direct conclusions about the efficiency of the ciphers based on the published papers. Incomparability stems from the different implementation assumptions, API's, hardware (e.g., processors) and software (e.g., compilers) used by implementers. Even more, some of the timings presented in previous papers correspond to "show-case" (as opposed to practically applicable) implementations, some examples of those being the fastest implementation of Twofish [SKW$^+$99b] that uses self-modifying code and Brian Gladman's implementations of AES candidates [Gla99] that use a number of key-specific static variables instead of allocating a register to address them, therefore effectively freeing some registers for other uses. Especially in the case of the Pentium family, where the number of available registers is very restricted, such implementations may result in a huge speed up. However, both types of implementation tricks restrict the application area of the implementation.

In the current paper we try to give a satisfactory answer to the question "what cipher is the fastest on the Pentium II" by carefully optimizing the 4 fastest AES candidates—MARS, RC6, Rijndael and Twofish—in Pentium II assembly, using for all implementations exactly the same, reasonable in practice, API and speed measurement conditions for all the ciphers. Due to this, our results are much fairer than most of the previously known ones: our implementations can be seen as black boxes applicable in almost any possible application of block ciphers on an environment featuring Pentium II. Additionally, careful optimization process resulted in implementations that are clearly faster than the previously known implementations. (Except for Twofish, which has still a faster "show-case" implementation.)

We start the paper by describing our platform of choice (Section 2), implementation philosophy and API (Section 3). Section 4 briefly surveys our results, and Section 5 gives more details on the problems encountered when implementing the ciphers. More information about the Pentium II is given in the Appendices.

2 Choice of the Platform

Our first principal choice was the decision what processor to use. By purely pragmatic reasons we decided that the implementation environment equips an Intel Pentium family CPU: while this family is not the most modern processor family available, it is the most widespread one at the moment of writing this paper and most probably also during the next few years. Therefore, since in the foreseeable future most of the software-based commercial security applications run on the Pentium family (as recognized also by the AES finalists designers), this family has the most direct impact on the choice of a cipher by security consumers.

At second, from the Pentium family we decided to choose the Pentium II processor. At first, it is a more advanced processor than Pentium Pro, the NIST AES analysis platform: the Pentium II provides (twice) larger register space due to the added MMX technology, and many new MMX-specific commands. Compared to the Pentium Pro, the Pentium II is also easier to obtain at the current stage, since Pentium Pro has been out of the manufacturing for a while. On the other hand, the Pentium II was preferred by the authors to the Pentium III since the latter is somewhat too new and controversial due to the privacy issues.

Another reason to choose Pentium II was that as the successor of the NIST AES analysis platform, implementing the AES candidates on the Pentium II could provide some insights on how generally suitable are the candidates, some of which were specifically optimized for the Pentium Pro, on future processors having features unpredicted by algorithm designers. While this is not as crucial as withstanding the "future attacks", it still gives some ideas about the possible longevity of the cipher. (We clearly would not want the AES in 20 years to have the role the 3DES has today!)

As shown in [Lip98], the MMX technology can seriously speed up IDEA ([LM90], [LMM94]), one of the believably most secure block ciphers with 64-bit block size. As stated in [Lip98], this can be done since IDEA has its key attributes similar to those of multimedia applications, for which the MMX technology was originally created. An open question posed in [Lip98] was how much would the MMX technology help implementing other ciphers, including the AES candidates. In the following we will partially answer to that question, showing that also some ciphers using only "simple" operations can greatly benefit from the added MMX technology. A short overview of Pentium II that is necessary for implementers and for cryptographers who design ciphers optimized for this platform is given in Appendix A. We refer for Intel manuals for a more complete overview.

3 Implementation Considerations

Several papers (including, in particular, [Lip99,SKW$^+$99a]) have compared AES candidates speed, but since the implementations quoted in them are often incomparable (or based on pure estimations), one cannot make direct conclusions about the efficiency of these algorithms based on the published papers. Incomparability stems from the different implementation assumptions, API's, hardware (processors) and software (compilers) platforms used by implementers. Even more, some of the numbers there correspond to the "show-case" (as opposed to practically applicable) implementations; including the bizarre case that one candidate was claimed to be the fastest on its inventors laptop under some suitable conditions.

As another example of the unsuitability of some "show-case" implementations, the fastest implementation of Twofish [SKW$^+$99b] uses self-modifying code and therefore cannot be used in a number of applications, while Brian Gladman's implementations of

AES candidates [Gla99] use a number of key-specific static variables instead of allocating a register to address them, therefore effectively freeing some registers for other uses. Especially in the case of the Pentium family, where the number of available registers is very restricted, such implementations may result in a huge speed up. On the other hand, Gladman's implementations cannot be used several applications, including multithreaded programs and SMP (symmetric multi-processing) systems.

Most of the security customers need however speed numbers applicable in whatever product they use in whatever environment in runs (for example, in a Linux kernel-supported IPSEC implementation, secure login or multithreaded access to encrypted storage arrays). For users it is necessary to know in what environment the measured speed numbers were obtained, to be able to calculate the possible efficiency of the ciphers in their own environments. Additionally, full specification is important for other implementers to be able to compare their implementations with ours. Hence, apart from providing "clean" implementations under some reasonable public assumptions, we shall also next fully specify these assumptions:

- We do not use self-modifying code ("code compilation" [SKW$^+$99b]) since it makes the implementation inapplicable in a number of situations, e.g., in operation-system kernel and ROM-based applications.
- We additionally decided not to use key-specific static areas since then the implementation could not be used, e.g., in SMP-capable systems and multithreaded programs.
- We decided to maximally use the MMX technology since it should not be forbidden in any reasonable modern environment. (While using self-modifying code and key-specific static areas is generally considered to be a bad programming practice.)
- We decided to use exactly the same API (specified later in Section 3.1) in all our implementations.
- A number of well-understood assumptions that 1) improve the speed and can be easily followed by implementers or 2) are essential to even be able to measure the speed:
  - All codes and data are correctly aligned.
  - Input and output texts and codes are preloaded to L1 cache in the possible extent to reduce the number of cache misses.
  - Simplicity of code: we tried to reduce time spent during writing and optimizing the code. In particular, all our implementations use highly optimized but round-number independent round macros. (Hence, our results could be slightly bettered if every round would optimized separately to avoid, e.g., delays in fetching stage.)

3.1 API

Since a different API can be influence the speed of an implementation severely, we also decided to fully specify the API used by us to make for the other implementers easier to compare their implementations to the ours. We felt that this is necessary, since AES candidate implementations reported in [Lip99] vary greatly in their API's.

```
void xxKS(char *master, uint32 bitLen, char *eKey);
void xxEnc(char *inBlk, uint32 lenBlk, char *eKey,
           char *outBlk);
void xxDec(char *inBlk, uint32 lenBlk, char *eKey,
           char *outBlk);
```

where

`xx` is algorithm name (e.g., `Rijndael`).

`xxKS` is key scheduling subroutine.

`xxEnc` is encryption subroutine that encrypts `lenBlk` blocks of plaintext starting from the address `inBlk` to the ciphertext location `outBlk`, by using extended key `eKey`, in ECB block cipher mode.

`xxDec` is decryption subroutine with the same input conventions as `xxEnc`.

`uint32` is the type of 32-bit unsigned integers (in the case of Pentium II, equal to `unsigned long` in the case of most compilers).

`master` is pointer to the master key bits.

`bitLen` is the bit length of a master key.

`eKey` is pointer to subkeys and other initialization data, used later by encryption and decryption.

`inBlk` is pointer to input texts to be encrypted in the case of `xxEnc` and to be decrypted in the case of `xxDec`.

`outBlk` is pointer to the corresponding output texts.

`lenBlk` is number of blocks to be encrypted or decrypted.

Fig. 1. Specification of our API.

Note that our API, depicted in Figure 1, is essentially equivalent to the API's used in most of the commercial applications, specifying only those inputs and outputs to the algorithms that are really needed by the algorithms. (Names of the subroutines and their parameters of course do not affect the speed, of course.) Our API was fixed for the key length of 128-bits due to the feeling that at the time when greater key sizes become necessary, our implementation platform would already be a history.

Here, the key schedule and decryption subroutines are specified only for completeness. Since in the current paper we are not interested in the optimization of these subroutines, we almost do not mention decryption and key schedules hereafter.

3.2 How to Measure a Number of Cycles

Different time measurement methods may change the speed numbers quite dramatically. As in the case of the API's, we decided to use one, sensible published and *fully specified* convention (specified in Figure 2) for all the implementations. (Note that this wrapping corresponds almost exactly to the method specified in [Fog00], to which the reader is referred for a throughout explanation of the method.) The inputs and key of the cipher are generated randomly before the measurement begins, to prevent "optimization" for specific class of keys. The input variable `lenBlk` was chosen to be equal to 8000 so that the input and output texts would not fit in the L1 cache. Also, `time` is a work area of type `uint32`, used in later calculations.

```
movd mm0, dword ptr [time];  /* warm cache and set MMX state */
xor eax, eax;
cpuid;                       /* serialize instructions */
rdtsc;                       /* read time-stamp counter */
mov dword ptr [time], eax;   /* save counter */
xor eax, eax;
cpuid;                       /* serialize instructions */
/* xxEnc() or xxDec() */
xor eax, eax;
cpuid;                       /* serialize instructions */
rdtsc;                       /* read time-stamp counter */
sub dword ptr [time], eax;   /* compute the difference */
emms;                        /* empty MMX state */
```

Note that `time` is a 4 bytes work area.

Fig. 2. Time measurement code

```
   /* push all used registers */
   cmp dword ptr [lenBlk], 0;
   jz L1;
   align 16;
L0:
   dec dword ptr [lenBlk];
   jnz L0;
L1:
   /* pop these registers once more */
```

Fig. 3. Null function

Note that this method has some overhead, due to both high latency of the `rdtsc` instructions and also the overhead caused by looping instructions like `jnz` which are not formally part of the cipher itself. (Looping instructions can be seen as a part of the block cipher mode, however.) We measure this overhead by using the null function shown in Fig. 3 obtaining a value `nulltime`, and then we subtract it from the value of `time` obtained by measuring the speeds of different encryption/decryption procedures. Finally, this result is divided by the number of blocks encrypted. Intuitively, by using this method we obtain the number of cycles corresponding to unrolled implementation of the block cipher, or to the implementation where we only care about the time encrypting one block takes without adding any extra overhead. (Note that the subtracted overhead number was equal to ≈ 6 in the case $n = 8000$. One could easily add this number to those presented later to get the number of cycles *with* overhead.)

Chosen time measurement method is also reasonable in practice: when the value of `lenBlk` was chosen to be different, for most of the implementations (*including* the implementation of null cipher), the execution times increased by almost the same constant. Hence, the null cipher proved experimentally to be well-defined.

Cipher	Mbits/s on a 450 MHz Pentium II	Cycles per block	Best previous result	Speedup
Null cipher	—	6	—	—
RC6	258 Mbits/s	223	243 [Riv98]	8%
Rijndael	243 Mbits/s	237	320 [DR98]	26%
Twofish	204 Mbits/s	282	315 [SKW$^+$99b]	11%
MARS	188 Mbits/s	306	390 [BCD$^+$98]	22%

Table 1. Performance in clock cycles per block of output of four AES finalists. (Only encryption considered)

Finally, we did a loop of 500 times over the described measurements and then chose the smallest number for every cipher, since that corresponds most likely to the case where most of the data and code are in L1 cache and the branch prediction works successfully: i.e., to the bulk encryption speed of the cipher itself.

4 Implementation Results

From the five AES finalists, one (Serpent) is regarded as a very conservative design but at the same time also being clearly slower than the other AES finalists. Rest of the finalists have comparable timings on most of the modern computer platforms, where one of the ciphers is the fastest in one platform, and another one in another platform. Since also on the Pentium II processor, Serpent seems to be very slow by the published data, we decided postpone its implementation to the future and concentrate on the fast ciphers.

Timings, obtained by measuring the speed of implementations by following previously specified procedures are summarized in Table 1[1]. The numbers in the middle columns show how many cycles it takes to encrypt one 128-bit block by using the chosen cipher with a 128-bit key. These results indicate that the chosen four AES finalists are extremely fast. For comparison, the standard hash algorithm SHA-1 *hashes* a 512-bit block in 837 cycles (i.e., 13.1 cycles per byte) and DES and 3DES encrypt a 64-bit block respectively in 340 and 928 cycles (resp., 42.5 and 116 cycles per byte) [PRB98], while RC6 and Rijndael respectively encrypt a 128-bit block in 223 and 237 cycles (resp., 13.9 and 14.8 cycles per byte). However, note that the cited timings in [PRB98] were obtained on a plain Pentium and therefore could most probably be improved on the Pentium II.

Our results seem to indicate, that the speed difference between different ciphers is less than expected: as before, RC6 is still the fastest cipher on the Pentium II, but the difference between it and Rijndael has decreased seriously. Hence we hesitate to say that RC6 is the fastest cipher. However, based on the cited results, we can classify the ciphers to two groups: blastingly fast ciphers RC6 and Rijndael and somewhat slower, but still very fast ciphers Twofish and MARS.

[1] We also started to code the decryption routines, finishing RC6 decryption (209 cycles per block) and Twofish decryption (276 cycles per block).

However, one has to keep in mind that RC6 and MARS have design features that make them specifically efficient on the Pentium Pro (and its successors), while their performance seriously degrades on other processors [Lip99,SKW$^+$99a]. This is due to the use of complex instructions (32-bit multiplication and data-dependent rotation) that are cheap on the P6 family (Pentium Pro, Pentium II, Celeron, Xeon and Pentium III) but very expensive on most of the other platforms. Interestingly, also the next generation Pentium processor (code-named "Willamette", [Int00]) has latency 10 multiplication and latency 2 or 4 shifts, as compared to latency 4 multiplication and latency 1 shifts on the P6 family [Int00, Section 4.1.3]. Hence, RC6 and MARS would considerably slow down on the Willamette, the next generation Pentium family processor. On the other hand, Rijndael and Twofish are based on simple operations, and run equally well on all platforms. The speed ratio between Rijndael and Twofish seems be remain *almost* the same on the other platforms [Lip99] (namely, Rijndael being $5 \dots 25\%$ faster than Twofish).

Note that the speed up percents in Table 1 correspond to the achieved speed ups compared to the fastest "clean" implementations (i.e., those not using key-specific static data or self-modifying code). However, these percents do not always mean that our implementation techniques were exactly as much better. For example, the best previous implementation of Rijndael was done for the plain Pentium, but not for the Pentium Pro: a factor that may have negatively affected its performance. The best previous "clean" implementation of MARS was written in C, and therefore had also a relatively slow performance. However, our own C implementation of MARS is clearly faster than the one given in Table 1. In the case of Rijndael, most of the acceleration Rijndael is due to the efficient use MMX technology. In general, speed up comes mainly from better optimization (elaborated tradeoff between processor operating stages) and full usage of the Pentium II possibilities (i.e., the MMX technology).

To further clarify how does the Pentium II architecture impact the speed, Table 2 shows the detailed information of our implementations in encryption mode in the micro-operation level. Usage of the table is simple. For example, in the intersection point of "@round" row and "port 01" column in `TwofishEnc` table one would find 19. That means that there are 19 μoperations in the round function of `TwofishEnc` which will be executed on port 0 or port 1.

Interestingly, our implementations of MARS, Rijndael and Twofish all require approximately the same number of μoperations, while RC6 is about two times "better" in this category. On the other hand, RC6 is also the worst cipher to parallelize: while in Rijndael, more than 2.5 μoperations are executed per a cycle, RC6 can only mildly use the super-scalar parallelism of Pentium II. More cipher-specific comments will be given in the next.

5 Cipher-Specific Comments

5.1 MARS

In the case of MARS [BCD$^+$98], the speed difference between a carefully optimized C implementation (using a recent snapshot of the `gcc` compiler) and an optimized assembly language implementation is only about 11% on the Pentium II. The speedup

	port 0	port 1	port 01	port 2	port 3	port 4	total
MARS encryption							(1.87 μops/cycle)
prewhitening			5	8			13
forward mixing	16		77	32			125
@core ($\times 16$)	6		9	3			18
backward mixing	16		85	32			125
postwhitening		1	8	4	4	4	21
total	128	1	319	124	4	4	572
RC6 encryption							(1.47 μops/cycle)
prewhitening			2	7			9
@round ($\times 20$)	8		5	2			15
postwhitening		1	4	5	5	5	20
total	160	1	106	52	5	5	329
Rijndael encryption							(2.54 μops/cycle)
whitening		1	8	6			15
@round ($\times 9$)	4	1	34	19			58
last round	4	3	31	20	3	3	64
total	40	13	345	197	3	3	601
Twofish encryption							(2.11 μops/cycle)
prewhitening			5	8			13
first round	5		19	10			34
@round ($\times 15$)	6		19	10			35
postwhitening	2	1	8	4	4	4	23
total	97	1	317	172	4	4	595

Table 2. Number of μoperations in our implementations

comes mainly from a slightly more efficient allocation of the integer registers and some (minimal) usage of the MMX instructions in the assembly implementation. However, the MMX technology is only moderately useful for MARS, since the complex instructions performed in MARS (i.e., 32-bit multiplication, data-dependent rotation and S-box lookups) are not available for the MMX registers. Additionally, due to the high data-dependency there is very limited freedom in meaningfully rescheduling the instructions in MARS, which also means that one cannot avoid all the delays on all the processor operating stages.

Another drawback is that during MARS encryption, some execution ports are considerably more overloaded than others. Namely, more than 78% of μoperations go either to port 0 or 1. The most overloaded is port 0, since 128 μoperations go only to this port — including 16 multiplications and extensively used rotations.

5.2 RC6

From implementers point of view, problems arising when optimizing an RC6 implementation are similar to those arising when coding MARS in many aspects: both ciphers rely on the same complex instructions, have long critical paths and overloaded port 0. However, since RC6 uses multiplications even more extensively, it is even less parallelizable. Table 2 shows that our implementation includes 160 port 0 μoperations, which includes 40 multiplications with latency 4.

RC6 is a very Pentium II-friendly cipher, and it is very easy to code it even in the assembly language. It can also be very efficiently implemented in C: the speed difference between a C implementation and an assembly implementation is about 18%. (The difference is bigger than in the case of MARS since `gcc`, the test compiler, performs very poorly in translating the quadratic formulas of type $x \cdot (2x + 1)$ to the Pentium II assembly language.) It is straightforward to obtain an optimized assembly language implementation from the C implementation: one does not have many possibilities to reschedule the code.

5.3 Rijndael

As opposed to MARS and RC6, Rijndael [DR98] is not C-friendly (at least not `gcc`-friendly) in the sense that assembly implementation is about 44% slower than `gcc`-implementation of the same cipher. It is however mainly due to the inefficiency of the `gcc` compiler: our implementation of Rijndael makes very heavy use of the MMX technology, but also of 8-bit instructions provided by Pentium family. However, `gcc` cannot efficiently use either of these.

Rijndael can effectively use the MMX since Rijndael is based only on most simple imaginable operations (`load`, `xor`), all of which are supported by the MMX technology. Additionally, since Rijndael has large internal parallelism (at least four-times, but partially up to 16-times parallelism!), there is a large number of possibilities to reschedule its code. Our implementation was obtained by doing so in a way that all the delays in the different stages of the Pentium II operation would be minimized. The final result is very impressive for the Pentium II: it executes 2.54 μoperations per a cycle.

Not the last factor that makes Rijndael suitable for the Pentium II is the fact that almost exactly one third of the μoperations in our implementation of Rijndael go to port 2, while the remaining $2/3$ of μoperations go to ports 0 and 1. Due to this and parallelism we get that during the Rijndael encryption 3 μoperations could be executed in parallel almost all the time. However, this (not to mention other aspects like decoding and fetching delays) also makes 20 cycles per round a lower bound for Rijndael and shows that our result may be very close to the optimal one. To facilitate more efficient implementations, the Pentium II should feature three ALUs, two concurrent memory access ports and also more decoders and retirement units: features that are not cipher-specific and would improve the speed of most of the applications.

Finally, we measured the timings of r-round Rijndael for variable r without any additional fine-tuning: those implementations are unoptimized since they use the same round macros as the 10-round Rijndael without any additional effort to optimize them to reduce, say, fetching delays. In particular it turned out that 8-round Rijndael (essentially equivalent to the cipher Square [DKR97] from the implementers point of view) encrypts a block in 193 cycles. 192-bit Rijndael (12 rounds) took 286 cycles, and 256-bit Rijndael (14 rounds)—333 cycles. Note that since 12-round Rijndael is very similar to Crypton [Lim98], 286 cycles is also a (hopefully) close approximation for the speed of latter.

5.4 Twofish

Twofish is designed to be well-suited on multiple platforms, including also the Pentium II. From the implementers point of view it resembles Rijndael in many aspects, by using only simple instructions but also some large-scale components of the latter (e.g., MDS, to provide diffusion). Due to the use of low-level instructions, Twofish is also relatively slow in C compared to the assembly (the difference is about 37%).

Main difference for implementers between Rijndael and Twofish is the inclusion of the Pseudo-Hadamard Transformation that somehow complicates Rijndael's clear structure and makes it less parallelizable: while the number of μoperations in our implementation of Twofish is less than in our implementation of Rijndael, it turned out to be very difficult to use the MMX technology to optimize Twofish. Hence, Twofish is only moderately parallelizable, although the parallelism of our implementation (2.11 μoperations per cycle) is relatively good.

6 Conclusion and Work in Progress

We achieved the fastest implementations of four of the AES finalists on the Pentium II processor, obtaining speedup $8\% \dots 26\%$ compared to the previously known implementations. Since all implementations were coded by using the same sensible assumptions, they provide a more adequate efficiency comparison of the AES finalists than the previous papers. We demonstrated that MMX can be quite efficiently used to speedup Rijndael, but is only moderately useful for other ciphers. (However, our implementations depend on the availability of MMX technology to a lesser or greater extent and in general do not run on the Pentium Pro.) We provided full specification on our time-measurement conditions to simplify for the future implementers to compare their implementations to ours.

Our implementations are not the final: we continue optimizing them. Up-to-date results will be available at the AES efficiency table [Lip99].

References

[ABK98] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Flexible Block Cipher With Maximum Assurance. In *The First Advanced Encryption Standard Candidate Conference*, Ventura, California, USA, 20–22 August 1998.

[BCD$^+$98] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko Zunic. MARS — A Candidate Cipher for AES. Original paper and a tweak to it are available from `http://www.research.ibm.com/security/mars.html`, June 1998.

[DKR97] Joan Daemen, Lars Knudsen, and Vincent Rijmen. The Block Cipher Square. In Eli Biham, editor, *Fast Software Encryption '97*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165, Haifa, Israel, January 1997. Springer-Verlag.

[DR98] Joan Daemen and Vincent Rijmen. The Block Cipher Rijndael. In *Third Smart Card Research and Advanced Applications Conference Proceedings*, 1998. To appear.

[FIP77] FIPS. Data Encryption Standard. Technical report, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1977. FIPS 46.

[Fog00] Agner Fog. How to Optimize for the Pentium Microprocessors. Available from `http://www.agner.com/assem/`, 11 March 2000.

[Gla99] Brian Gladman. AES algorithm efficiency. Unpublished. Information available from `http://www.btinternet.com/˜brian.gladman/ cryptography_technology/`, January 1999.

[Int99] Intel. *Intel Architecture Optimization. Reference Manual*, 1999. Order Number 245127-001.

[Int00] Intel. *Willamette Processor Software Developer's Guide*, February 2000. Order Number 245355-001.

[Lim98] Chae Hoon Lim. Specification and Analysis of CRYPTON Version 1.0. Unpublished. Available from `http://crypt.future.co.kr/˜chlim/ pub/cryptonv10.ps`, 22 December 1998.

[Lip98] Helger Lipmaa. IDEA: A cipher for multimedia architectures? In Stafford Tavares and Henk Meijer, editors, *Selected Areas in Cryptography '98*, volume 1556 of *Lecture Notes in Computer Science*, pages 248–263, Kingston, Canada, 17–18 August 1998. Springer-Verlag.

[Lip99] Helger Lipmaa. AES candidates: A survey of implementations. An on-line table. Information available from `http://home.cyber.ee/helger/aes/`, January 1999.

[LM90] Xuejia Lai and James Massey. A proposal for a new block encryption standard. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer-Verlag, 1991, 21–24 May 1990.

[LMM94] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In D. W. Davies, editor, *Advances on Cryptology — EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38, Brighton, UK, April 1994. Springer-Verlag.

[PRB98] Bart Preneel, Vincent Rijmen, and Antoon Bosselaers. Recent developments in the design of conventional algorithms. In B. Preneel, R. Govaerts, and J. Vandewalle, editors, *Computer Security and Industrial Cryptography, State of the Art and Evolution*, volume 1528 of *Lecture Notes in Computer Science*, pages 90–115. Springer-Verlag, 1998.

[Riv98] Ronald L. Rivest. Futher Notes on RC6. Unpublished. Available from `http://theory.lcs.mit.edu/˜rivest/rc6-notes.txt`, 20 June 1998.

[RRSY98] Ronald L. Rivest, Matt J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 Block Cipher. Available from `http://theory.lcs.mit.edu/˜rivest/rc6.ps`, June 1998.

[SKW$^+$99a] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. Performance comparison of the AES submissions. Unpublished. Information available from `http://www.counterpane.com/`, January 1999.

[SKW$^+$99b] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*. John Wiley & Sons, April 1999. ISBN: 0471353817.

A Pentium II for Cipher Designers and Implementers

A.1 MMX Technology

The Pentium II has 8 integer (including stack pointer) and 8 new MMX registers; the latter were not present in the Pentium Pro. While there is a great number of operations available on the integer registers, MMX registers are much more “RISCy”: only a few instructions affect them, including move, Boolean operations, 16-bit arithmetic and shifts. Available set of instructions does not include several operations used in the modern block cipher design, including rotation and 32-bit multiplication. On the other hand, the MMX technology provides 64-bit versions of Boolean operations and data moves (i.e., the simplest possible operations), and also parallel 4-way addition and multiplication of 16-bit data. 16-bit multiplication is currently used in a very few ciphers, but as shown in [Lip98], ciphers that base their security on extensive use of 16-bit multiplication can be speed up considerably if using the MMX technology.

Despite of MMX’s attractiveness, at the current state of the affairs many C compilers (for example, `gcc`, the standard compiler for Linux machines) do not yet produce MMX code. Hence, for the Pentium II the assembly implementations are potentially more efficient than C-language implementations. Partially by this reason, many designers and implementers of AES candidates seem not to know about MMX at all.

A.2 Processor stages.

The Pentium II processor (as other processors in the P6 family) operates in several stages. At first the instructions are fetched from the main memory and then broken down (decoded) into μoperations (simple instructions consist of only one μoperation, while complex instruction have more μoperations). Thereafter, the μoperations go via a short queue to the register allocation table that allows register renaming. After that, instructions go to reorder buffer that enables out-of-order execution. There it stays until the operands it needs are available. Ready-for-execution μoperations are sent to the execution units, and thereafter retired [Int99,Fog00]. During the optimization one has to count on all different stages of processor operation to find a good tradeoff between the delays introduced in them. The technicalities presented hereafter could be most interesting for the implementers, but also for the cipher designers who want to create ciphers optimized for the Pentium II. The most important lesson from the next is that fixing any processor stages (e.g., decoding), suitable reordering of the instructions can considerably reduce the delays at this stage. However, the same reordering usually introduces additional delays in some other stages and therefore, code reordering is always a complicated tradeoff. To achieve really fast implementations, a cipher should have great internal parallelism that provides many different instruction reordering possibilities, from what the best could be found after possibly exhaustive search. Of course, one could design a cipher that would have only one possible order of instructions, optimized specifically for Pentium II. However, such cipher could slow down severely if even slightest modifications would be introduced to the processor. Moreover, parallelism is necessary anyways, since already in the near future a processor could have dozens simultaneously working executing units.

Note that our survey is far from being complete, we refer an interested reader to [Int99,Fog00]. However, during finishing our implementations we found that also the official Pentium family optimization manual published by Intel [Int99] is far from being complete. We encountered many problems that could not have been foreseen by using only the official manuals. Often more accurate (although also not complete) information about the Pentium II was found in [Fog00]. In several places of our implementations we performed partial exhaustive search to optimally schedule the instructions. A lot of experience and luck is necessary in optimizing for Pentium II if one desires to avoid exhaustive search himself.

In-Order Decoding. Up to 3 instructions can be decoded to μoperations at time, but only the first decoder can handle instructions with more than one μoperation. It is recommended to order the instructions in the 4-1-1 sequence, which means that only every third instruction could combine in itself of more than one μoperation [Int99]. By this reason, algorithms using only "simple operations" can be potentially implemented faster than those consisting of "complex instructions". However, in some circuimstances it would also beneficial to have at least some complex instructions. Namely, if the code is properly scheduled in a way that exactly (almost) every third instruction has more than one μoperation, the decoder will feed the out-of-order execution pool with pace more than 3 μoperations per cycle. Now, if in some later stage less than 3 μoperations per cycle are fed to the execution unit (say due to the delays in fetching), this unit will not idle waiting for the next instructions from the decoder.

Instruction In-Order Fetching. The Pentium II has 16-byte internal *ifetch buffers* with the peculiarity that a new buffer is forced to start at beginning of an instruction. The first instruction of the ifetch buffer will be always decoded by decoder 0, even if the previous instruction was decoded by the same decoder and hence, other decoders would stay idle. Hence, code reordering and possible use of semantically identical instructions (in general, but not always, *shorter* instructions: for example, `mov eax,[ebx+0]` with `mov eax,[ebx]`) with different length could reduce the number of delays introduced in this stage.

Register In-Order Renaming. Pentium II has 40 hardware registers. The software registers are renamed to hardware registers after a write to (or read from) the software register. After a register has not been used for a while, it automatically retires and the next time the same register is used, a new renaming is performed. It is important to know that *only two register renamings can be done during one machine cycle*. In particular this means that generally it is beneficial to gather all instructions operating on some fixed data chunk together (i.e., to reorder the code in a suitable way). However, it is extremely difficult to detect and remove delays introduced by this stage, and therefore this stage may really become *the* bottleneck in optimization: subtle modification of code may introduce long delays in this stage. We refer to [Fog00] for more information.

Out-of-Order Execution. Pentium II has 5 execution ports (port 0, port 1, $\dots$, port 4) that can execute instructions out-of-order. Every port has some specific meaning.

Ports 0 and 1 are ALUs (they can perform arithmetic on operands in registers), port 2 performs memory loads. Every memory write counts as two μoperations, one in port 3 (address calculation) and another one in port 4 (memory write). Up to 3 ports can execute an instruction in parallel. There are a number of arithmetic instructions that can only run in port 0 (most importantly, multiplication, rotation and integer register shifts — instructions that are widely used by some AES finalists), while some other instructions (most importantly, MMX register shifts) can only run in port 1. To obtain a throughput near to 3 μoperations per cycle, the instructions should be distributed so that no more than $2/3$ of them are arithmetic, no more than $1/3$ are memory loads and no more than $1/3$ are memory writes: a condition that is very difficult to fulfill in a practical application.

In-Order Retirement After execution, μoperations will retire in-order. During retirement, hardware registers will be written back to software registers and the μoperations leave the instruction pool. Since this is done in-order, several delays can occur, e.g., if speculative out-of-order execution of some earlier long latency instruction is not finished at the moment of retirement.

Session 3:

"Surveys"

A Performance Comparison of the Five AES Finalists

Bruce Schneier
Counterpane Internet Security, Inc.
3031 Tisch Way, Suite 100PE
San Jose, CA 95128

Doug Whiting
Hi/fn, Inc.
5973 Avenida Encinas, Suite 110
Carlsbad, CA 92008

15 March 2000

1 Introduction

In 1997, NIST announced a program to develop and choose an Advanced Encryption Standard (AES) to replace the aging Data Encryption Standard (DES) [NIS97a,NIST97b]. They solicited algorithms from the cryptographic community, with the intent of choosing a single standard. Fifteen algorithms were submitted to NIST in 1998, and NIST chose five finalists in 1999 [NBD+99]. NIST's plans to choose one (or more than one) algorithm to become the standard in 2000.

NIST's three selection criteria are security, performance, and flexibility. This paper is primarily about the latter two criteria, in light of information regarding the first criterion. In Section 3, we discuss the software performance of the five AES finalists on a variety of CPUs. Detailed information can be found in [SKW+99b], and is not repeated here.

Specifically, we compare the performance of the five AES finalists on a variety of common software platforms: current 32-bit CPUs (both large microprocessors and smaller, smart card and embedded microprocessors) and high-end 64-bit CPUs. Our intent is to show roughly how the algorithms' speeds compare across a variety of CPUs.

There has been considerable discussion in the community about taking the submissions and modifying the number of rounds to increase security. The effect of this is that successful cryptanalytic attacks against a submission do not necessarily knock it out of the running, but instead decrease its performance (by forcing the number of rounds to increase). In [Bih98,Bih99], the author suggests comparing "minimal secure variants" as a way to normalize algorithms.

In Section 4, we give the maximum rounds cryptanalyzed for each of the algorithms, and re-examine all the performance numbers for these variants. We believe that this provides a fairer way of comparing the different algorithms, and the design decisions the different teams made. We then compare the algorithms again, using the minimal secure variants as a way to more fairly align the security of the five algorithms.

2 Performance as a Function of Key Length

The speed of MARS, RC6, and Serpent are independent of key length. That is, the time required to set up a key and encrypt a block of text is the same, regardless of whether the key is 128, 192, or 256 bits long. Twofish encrypts and decrypts at a speed independent of key length, but takes longer to set up longer keys.[1] Rijndael both encrypts and decrypts more slowly for longer keys, and takes longer to set up longer keys. The results are summarized in Table 1.

Algorithm Name	Key Setup	Encryption
MARS [BCD+98]	constant	constant
RC6 [RRS+98]	constant	constant
Rijndael [DR98]	increasing	128: 10 rounds 192: 20% slower 256: 40% slower
Serpent [ABK98]	constant	constant
Twofish [SKW+98, SKW+99a]	increasing	constant

Table 1: Speed of AES Candidates for Different Key Lengths

[1] Twofish is more flexible than this chart implies. There are implementations where the encryption speed is different for different key lengths, but they are only suitable for encrypting small text blocks [SKW+98a,WS98,SKW+99a].

In our first performance comparison [SKW+99b], we concentrated on key setup and encryption for 128-bit keys. In this paper, we look at all three key lengths.

3 Software Performance

Efficiency on 32-bit CPUs is one of NIST's stated performance criteria. Unfortunately, modern architecture of modern CPUs is so complicated that there is no single performance number that can be used as a basis for comparison.

Today, most high-end microprocessors use 32-bit architectures. These microprocessors range from the Intel Pentium family to embedded CPUs like the ARM family to 32-bit smart card chips. Since all of the AES finalists use 32-bit word sizes, it is not surprising that they are most efficient on these architectures.

The performance space covered by 32-bit CPUs is actually very large, from a 386 or a 68000 or an embedded RISC CPU on the low end, through the various CPUs in the Pentium family. Each CPU chip has its own set of strengths and weaknesses, including clock frequency, cache size and architecture, instruction pipelining (scalar vs. superscalar), etc. Each feature can have significant impact on the speed with which an algorithm can be executed. Even within the same family, major differences in the relative efficiency of certain instruction types can be manifest. As a particular case in point, the Pentium CPU is relatively quite slow at performing multiplies and variable rotates compared to the Pentium Pro/II/III CPUs. This is of interest because two of the AES candidates rely heavily on such operations. It would seem encouraging that the trend is toward better performance on these opcodes in later generations, but in fact future generation processors (such as the IA-64 family) seem to revert back to relatively slow performance for these instructions. The point here is that relative performance of the AES algorithms may vary dramatically on various CPUs. For example, RC6 is the fastest algorithm on the Pentium II/III family by a small margin, but is less half the speed of the fastest candidates on the Pentium and the PA-RISC (and, reportedly, on the Intel Itanium, whose architecture is heavily influenced by PA-RISC). It is not clear exactly what conclusion to draw from such anomalies, particularly since all the candidates are quite fast compared to Triple-DES. In any case, great care should be taken not to assign rankings based solely on a single generation of a single CPU family, which may have a somewhat idiosyncratic performance.

Over the next several years, 64-bit CPUs will become the default microprocessor in desktop computers. These include the Intel Itanium (formerly the Merced) and McKinley, and the DEC Alpha. These microprocessors will first be designed into high-end servers and workstations, and will eventually migrate to commodity desktop computers.

3.1 Language Choice

The language of implementation should not matter in theory; i.e., the relative performance of the algorithms in various languages should not depend on the language. In practice, this is at least partially true. For example, some algorithms are very "compiler friendly" and perform quite nicely in high-level languages. To some extent, this is dependent on the language itself, such as the lack of a rotation primitive in C, but to a larger extent it depends on the compiler. The ratio between speeds of Serpent in C and assembler seems to be close to 1:1 on most platforms, while for other algorithms it is between 1.5:1 and 2:1. Thus, Serpent looks relatively much better in C than in assembler. The Java performance seems frankly so dependent on the compiler that the mere phrase "Java performance" seems like an oxymoron. Ultimately, the choice of language may shift the relative performance numbers across algorithms by up to a 2:1 ratio. As long as the metric of choice is somewhat independent to such a range, language really doesn't matter. However, if the ultimate 10% or 50% performance really matters, the algorithm of choice will always be written in assembly—encryption is a discrete and well-defined chunk of code with a single entry point that needs to run fast, a perfect example of something that should be coded in assembly—so those are the best numbers to use in attempting to get precise comparisons.

3.2 Comparing Software Performance

The tables in this section compare the five AES finalists on different CPUs. We either used the data in the candidate's AES submission documentation or, where such data was unavailable or unreliable, calculated our own or used the data from other people's implementations [Gla98,Alm99,BGG+99,Gra00,Lip00]. Where we were unsure if a particular optimization technique would work, we gave the algorithm the benefit of the doubt. When there were multiple sources of data, we took the fastest. We believe this is a fair comparison of the algorithms' speeds.

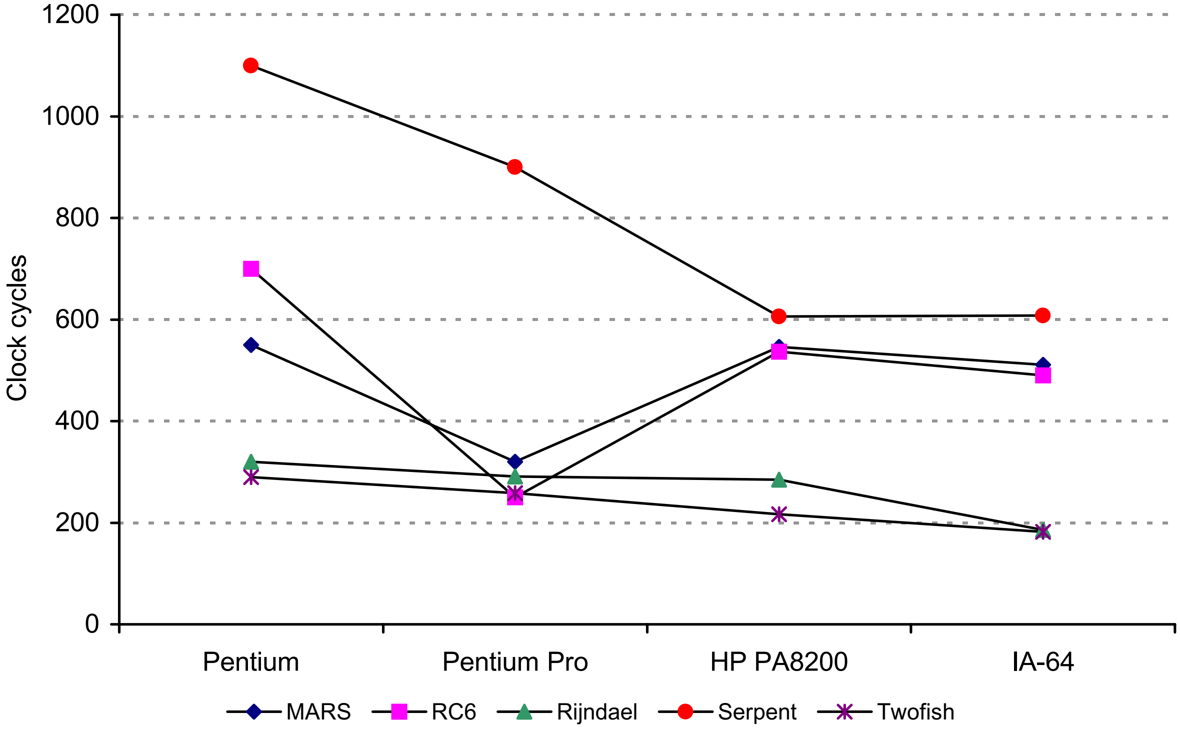


Figure 1: Encryption Speeds for 128-bit Keys in Assembly

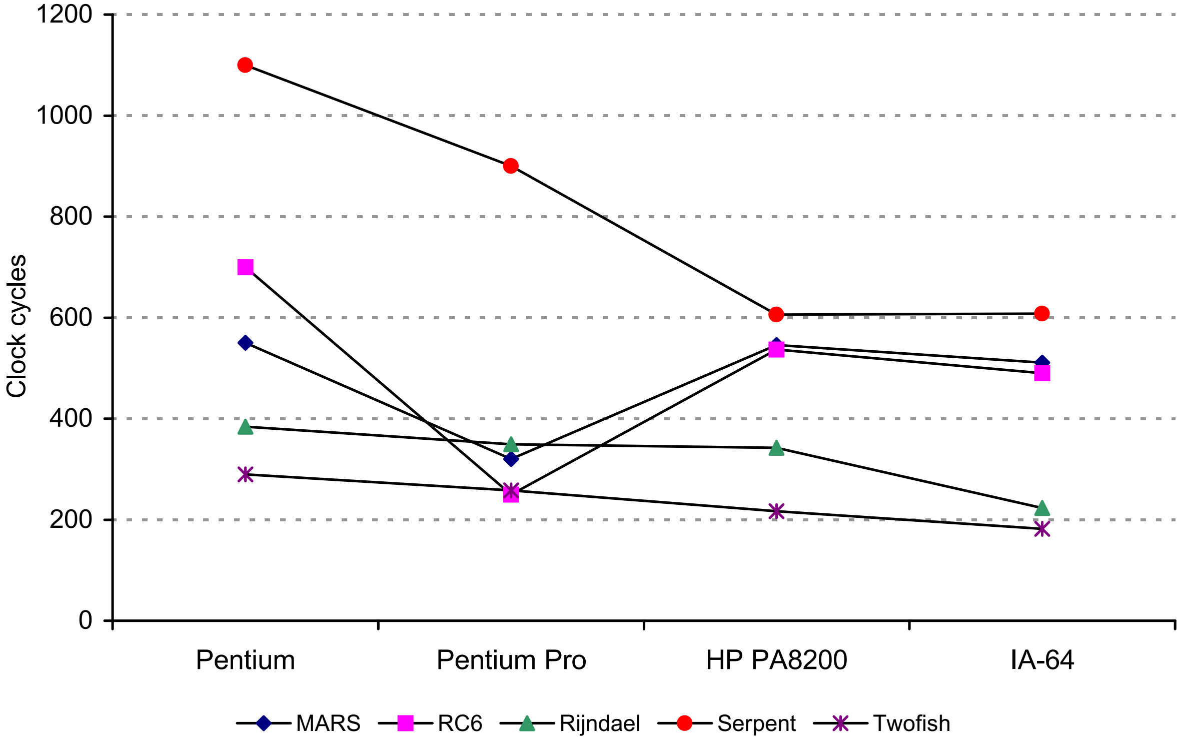


Figure 2: Encryption Speeds for 192-bit Keys in Assembly

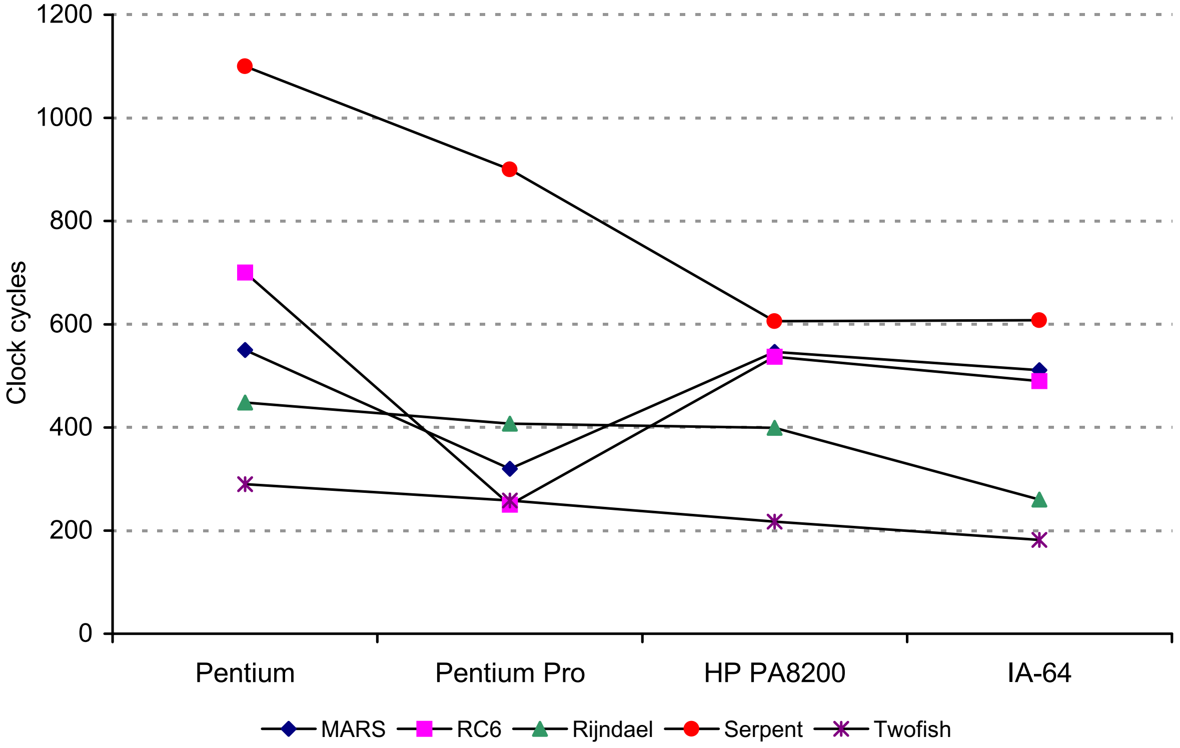


Figure 3: Encryption Speeds for 256-bit Keys in Assembly

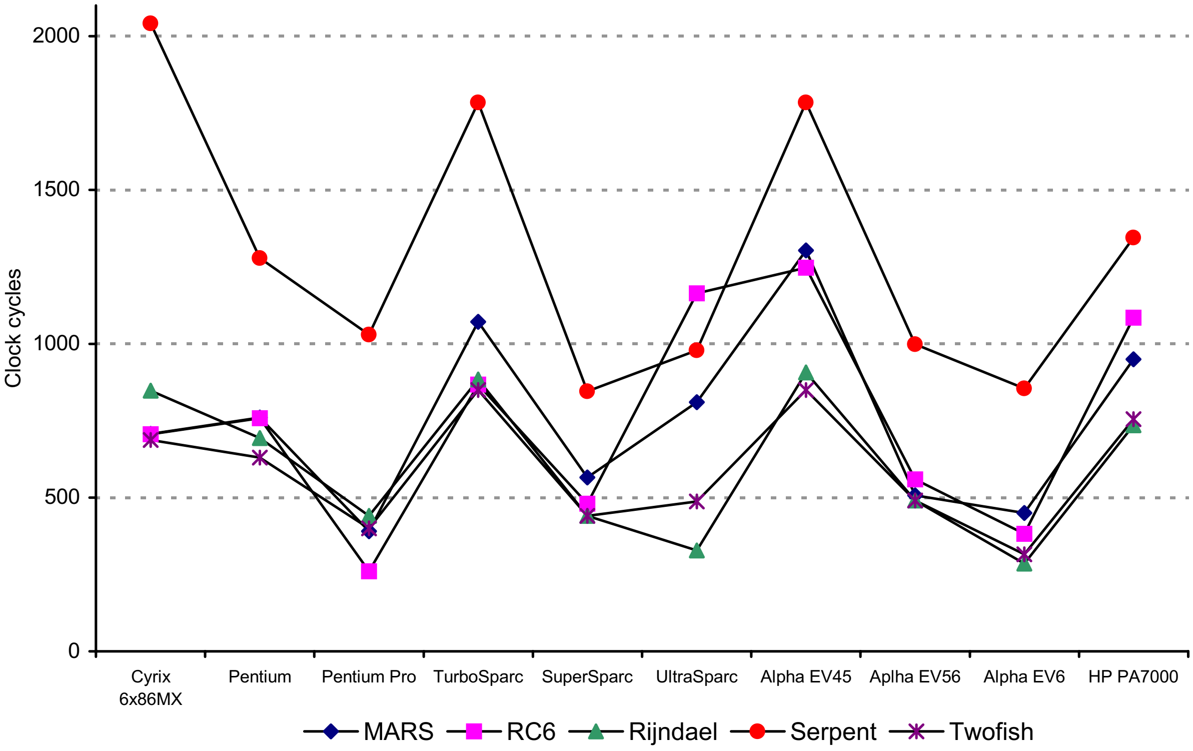


Figure 4: Encryption Speeds for 128-bit Keys in C

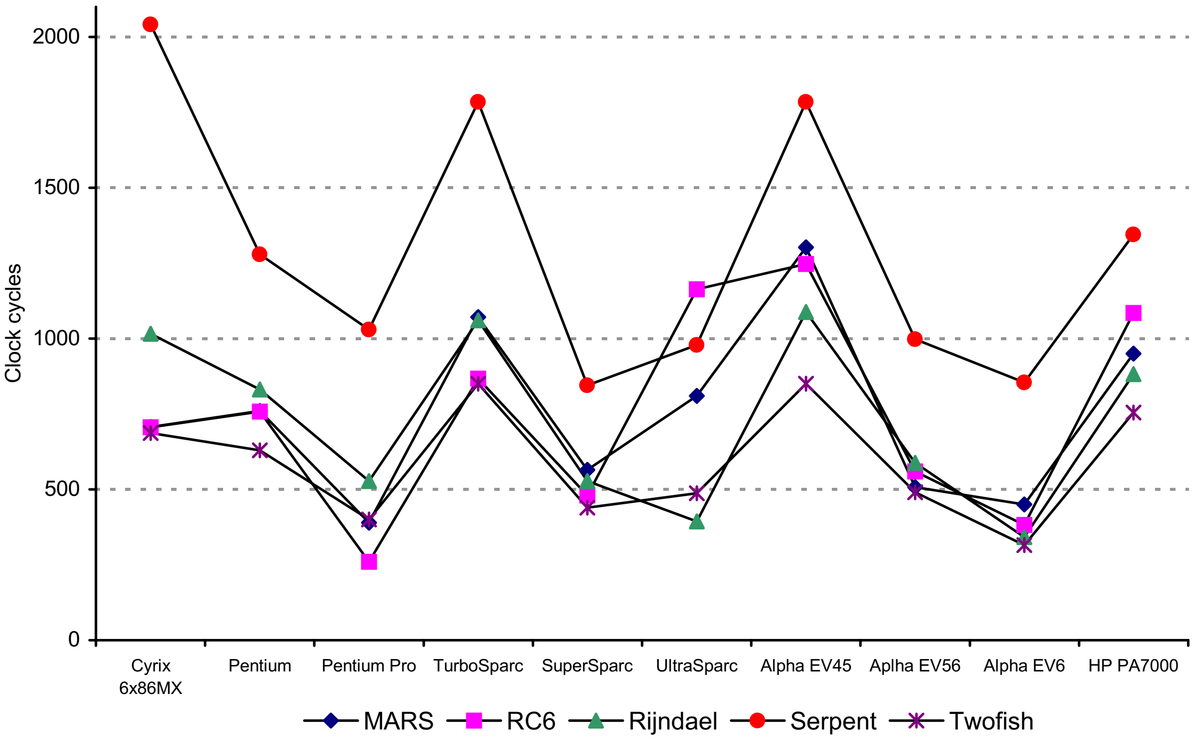


Figure 5: Encryption Speeds for 192-bit Keys in C

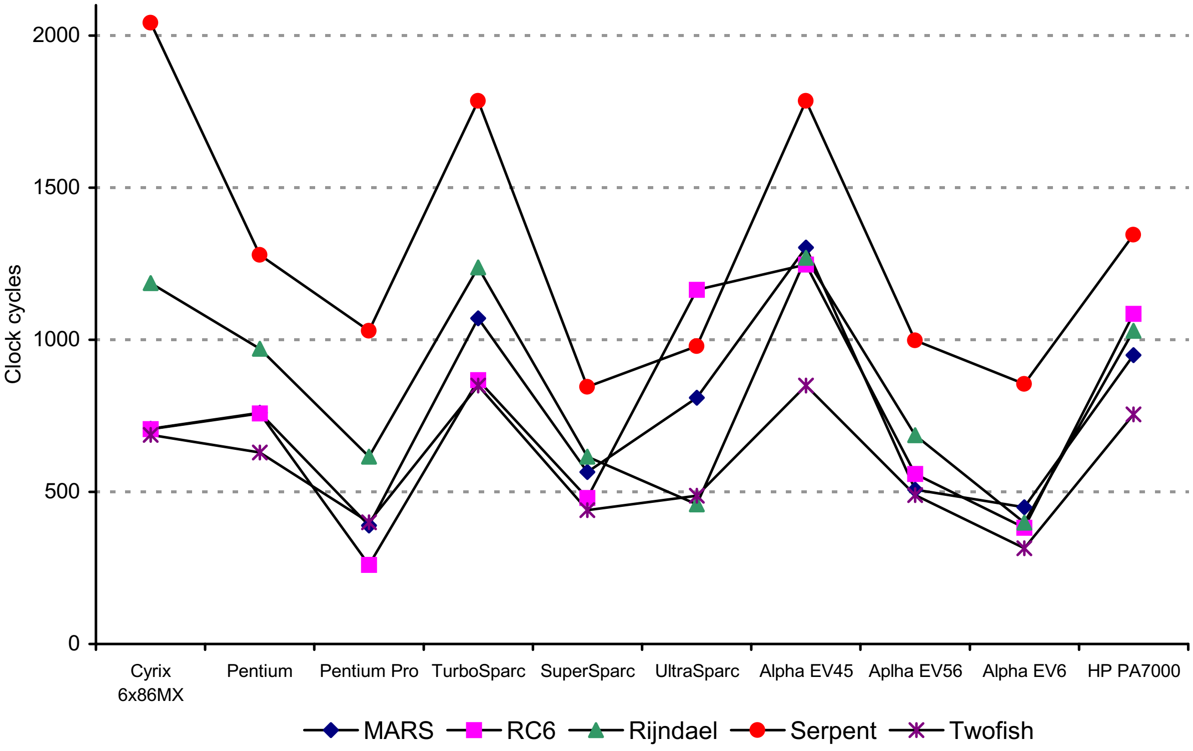


Figure 6: Encryption Speeds for 256-bit Keys in C

Interestingly enough, with the exception of MARS and RC6 on Pentium-Pro–class CPUs, the relative performance of the different algorithms was fairly constant. MARS and RC6 are exceptions because they rely heavily on data-dependent rotations and 32-bit multiplications. These operations are not part of the standard RISC instruction-set core, and their performance varies heavily with CPU [SW97]. Normally these two operations are very slow, but on the Pentium Pro, Pentium II, and Pentium III they are fast. Hence, these two algorithms are relatively faster on these CPUs than on both more and less sophisticated CPUs.

For 128-bit keys, Rijndael and Twofish are the fastest algorithms, MARS and RC6 are in the middle, and Serpent is slowest. For larger key lengths, Rijndael gets progressively slower; in some implementations it becomes slower than MARS. These trends hold true in both C and assembly language, although Serpent tends to produce C code that is closer to its assembly-language performance than the others.

3.2.1 Key Setup Plus Encryption

For encryption of short blocks of plaintext, performance is a function of both encryption speed and key setup speed. Much less data is available on key setup on various platforms, but we can estimate from the data we have. We use the key setup estimates from [SKW+98b]. Note that the numbers for Twofish take advantage of its flexibility in key setup vs. encryption.

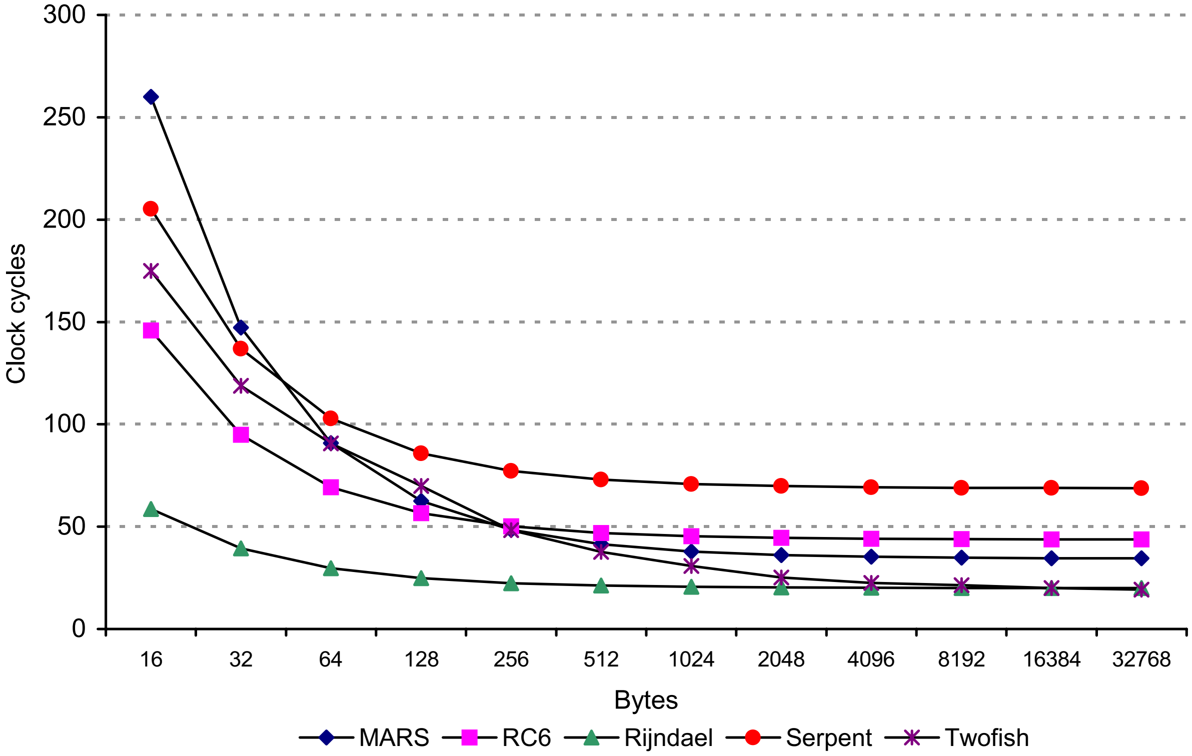


Figure 7: Key Setup and Encryption Rate, per Byte, for 128-bit Keys on a Pentium in Assembly

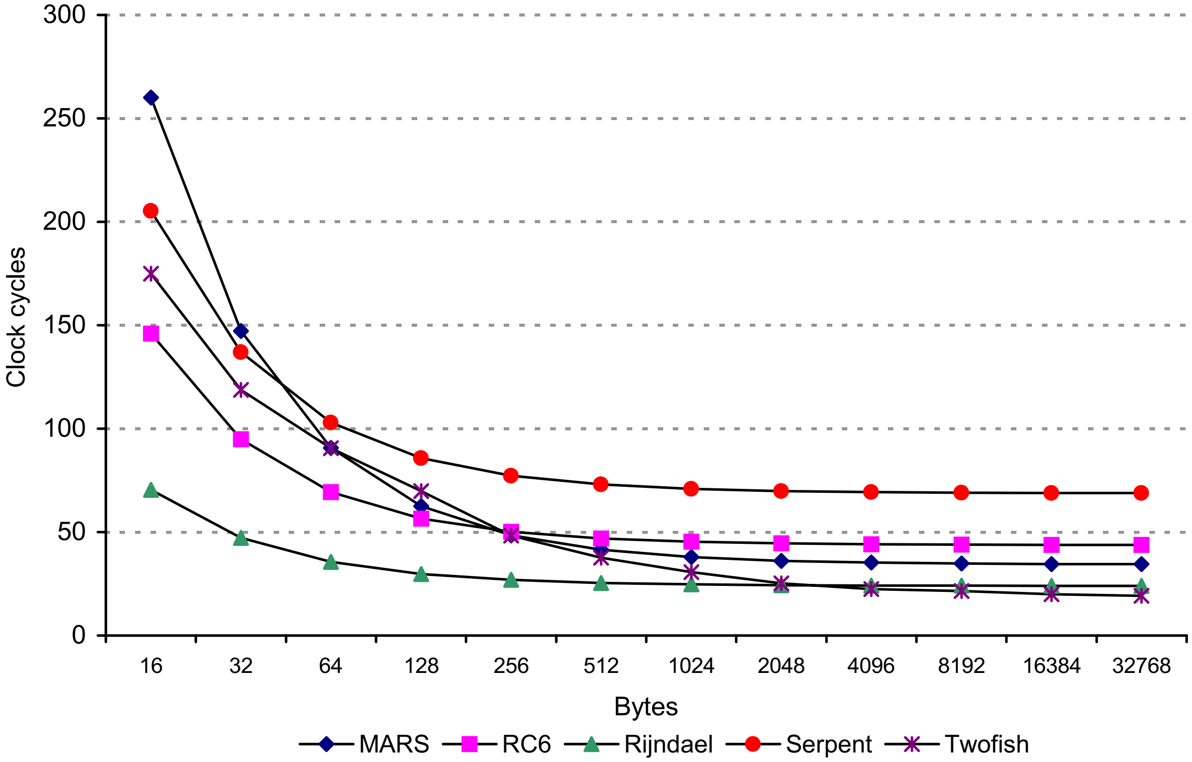


Figure 8: Key Setup and Encryption Rate, per Byte, for 192-bit Keys on a Pentium in Assembly

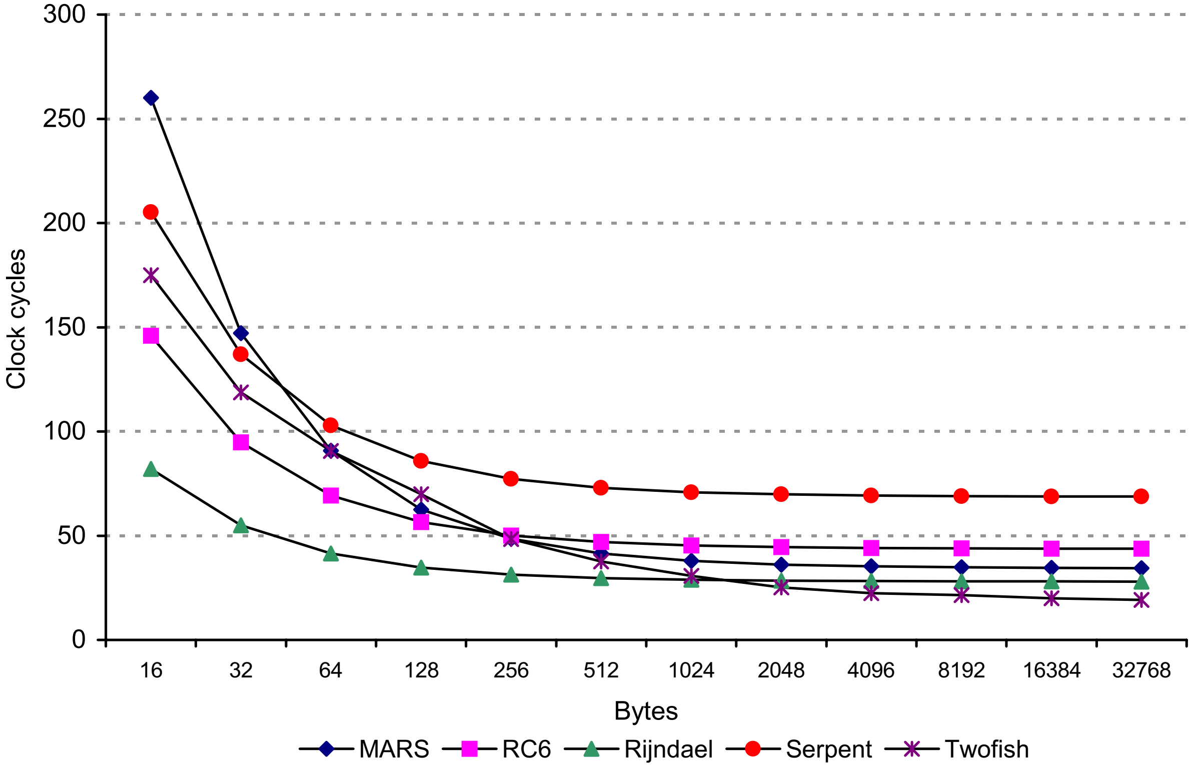


Figure 9: Key Setup and Encryption Rate, per Byte, for 256-bit Keys on a Pentium in Assembly

Rijndael is by far the best algorithm for encrypting small blocks, although the algorithms quickly normalize to their "natural" ordering. The 16-byte measure is of particular interest, because that is the speed of the algorithm if used as a hash function.

4 Software Performance of "Maximal Insecure Variants"

In [Bih98,Bih99], Biham introduced the notion of comparing the algorithms based on their "minimal secure variants." Different design teams were more or less conservative than each other; the number of rounds in their final submissions was not a fixed percentage of the number of rounds they could successfully cryptanalyze. Biham tried to normalize the algorithms by determining the minimal number of rounds that is secure (either as described by the designers or other cryptographers, or Biham's "best guess"), and then added a standard two cycles.

In his comments to NIST [Knu99], Lars Knudsen presented another rule of thumb for changing the number of rounds of the different algorithms: "Let *r* be the maximum number of rounds for which there is an attack faster than exhaustive key search. Choose *2r* rounds for the cipher." This rule gives us a new, although similar, measure of comparison.

For this comparison, we leave the scaling factor for another discussion, and simply compare the maximal number of rounds for which the best cryptanalytic attack is less complex than a 256-bit brute-force search; call this the "Maximal Insecure Variant." For Table 2, we use the best published cryptanalytic results as explained below.

Algorithm Name	Rounds
MARS	9 of 16
RC6	15 of 20
Rijndael	8 of 14
Serpent	9 of 32
Twofish	6 of 16

Table 2: Maximal Insecure Variants

If there is a weak key class, we count the attack if the probability of finding the weak key times the complexity of the attack is no more than 2^{256}. If there is a related-key attack, we make a similar calculation.

- MARS is complicated because there are four different types of round functions. The MARS design team believes that the cryptographic strength of the algorithm is in the "core"; hence, we concentrate on those rounds. There is an 11-round (of 16 total) attack of the MARS core [KKS00a]. There is also an attack against the cipher with the four different round functions symmetrically reduced from 8 rounds to 3 [KS00]. We make the somewhat arbitrary decision to take a scaling "halfway" between those two results: 9 rounds.
- RC6 has an attack against 15 rounds [KM00]. This attack also applies to a weak key class; the attack works for 1 in 2^{60} keys, and the complexity of the attack is 2^{170}. Hence, this attack counts by our definition. The RC6 designers estimate that 16 rounds is attackable, although they give no concrete attack.
- Rijndael has a distinguishing attack against 8 rounds [FKS+00a].
- Serpent has a distinguishing attack against 9 rounds [KKS00a,KKS00b]. The authors estimate that the longest variant that is not as secure as exhaustive search is 15 rounds, although they have no attack.
- Twofish has attacks against 6 rounds. The related-key attacks discussed in [SKW+98,SKW+99a] do not work [FKS+00b].

Unfortunately, these comparisons are fundamentally flawed, because they unfairly benefit algorithms that have been cryptanalyzed the least. Despite almost two years in the public, there has been little cryptanalysis of the five algorithms. We urge caution in reading too much into these numbers, and would like to see further cryptanalysis of the five algorithms.

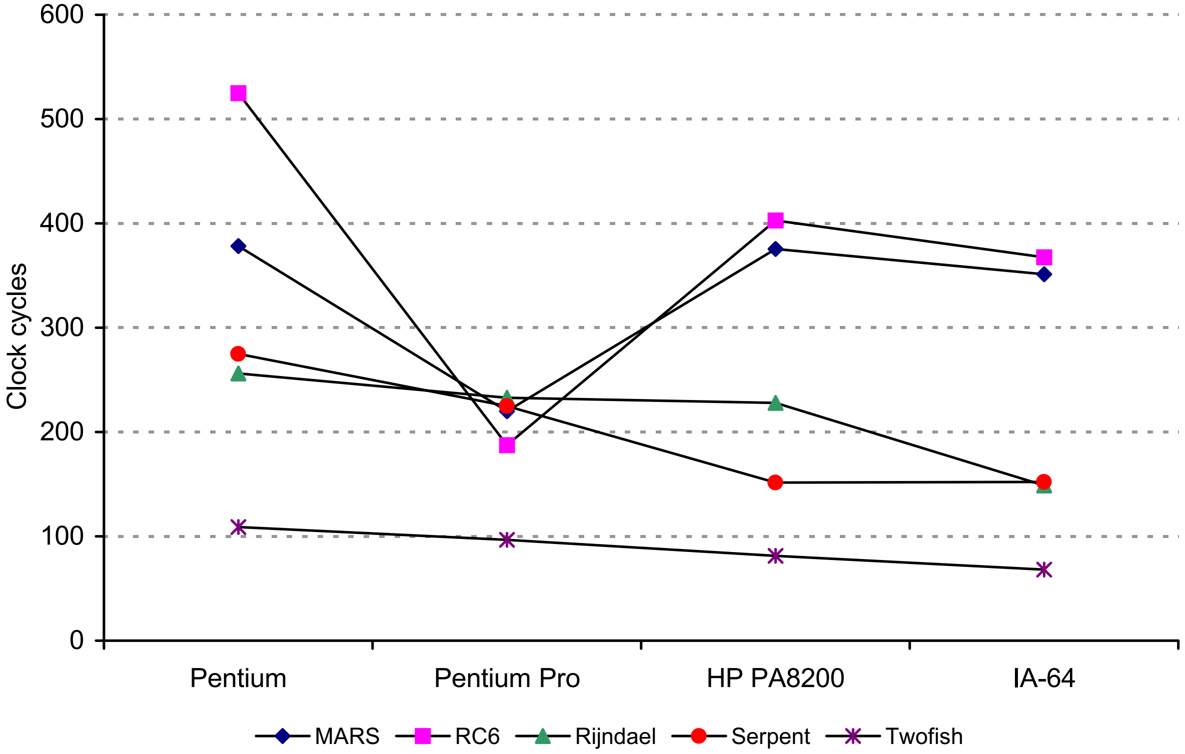


Figure 10: Encryption Speeds for the Minimal Secure Variant In Assembly

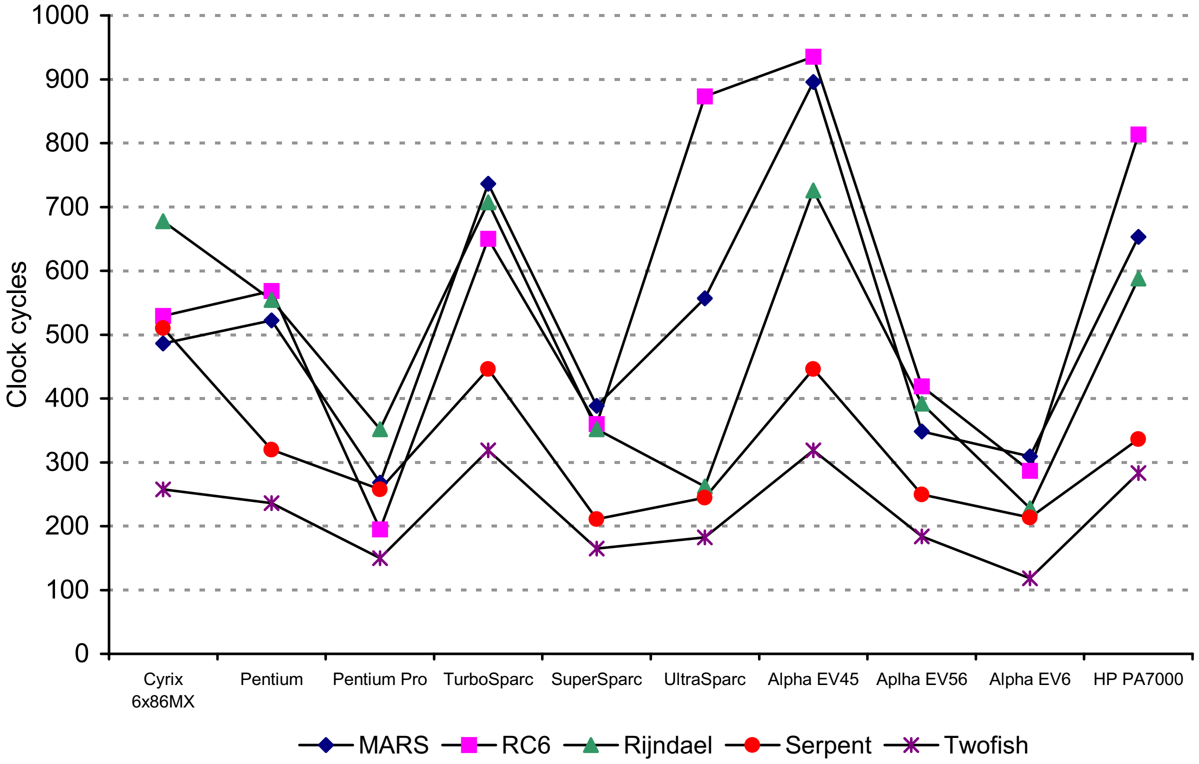


Figure 11: Encryption Speeds for the Minimal Secure Variant in C

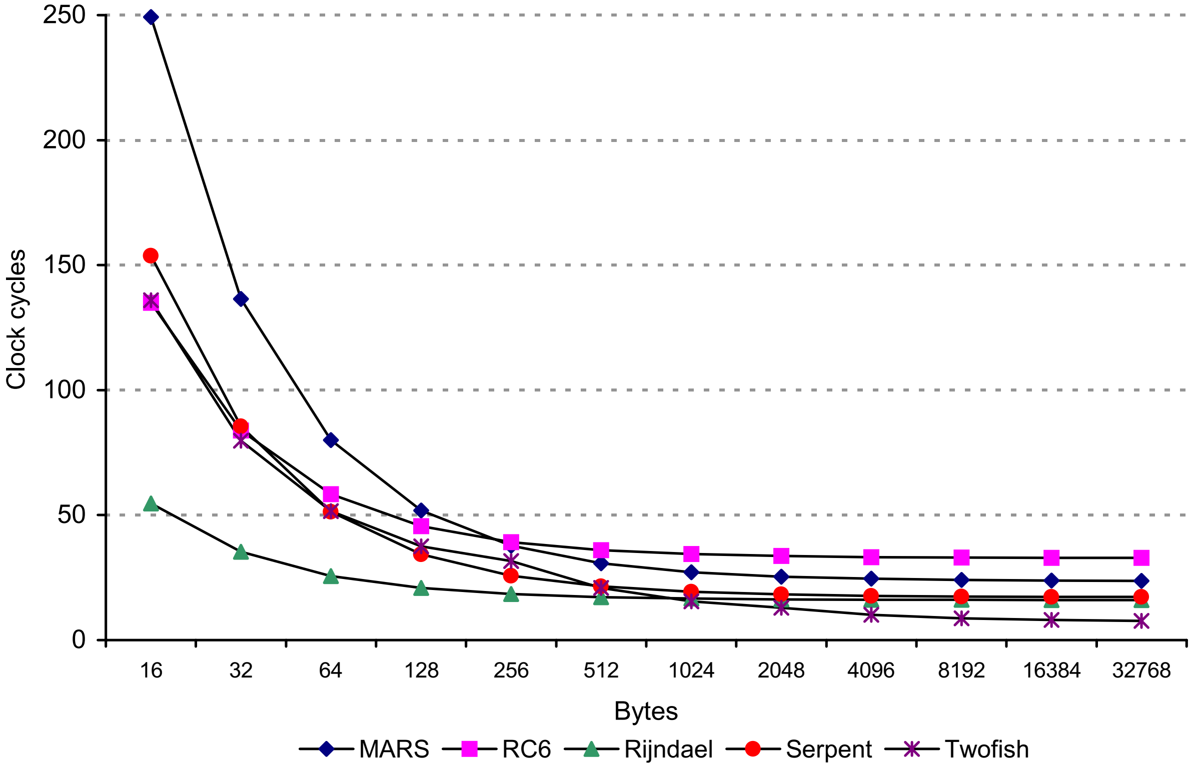


Figure 12: Key Setup and Encryption Rate, per Byte, for the Minimal Secure Variant on a Pentium in Assembly

Of course we are not recommending using these round numbers for the algorithms; any AES selection would obviously scale these numbers by a small linear factor. But the relative speeds would be independent of this scaling, and are hence relative to the selection process.

5 Performance on Memory-Limited 8-bit Smart Cards

As discussed in [SKW+99b], 8-bit implementations tend to be concerned more with fit than with performance. That is, RAM requirements are more important than clock speed.

Most commodity 8-bit smart card CPUs today include from 128 to 256 bytes of on-board RAM. Each CPU family typically contains members with more RAM and a correspondingly higher cost. RC6 has no effective way to compute subkeys on the fly, thus requiring that all the subkeys be precomputed whenever a key is changed. The RC6 subkeys consume from 176 bytes of "extra" RAM—more than is available on many commonly used CPUs. Although some CPUs include enough RAM to hold the subkeys, it is often unrealistic to assume that such a large fraction of the RAM can be dedicated solely to the encryption function. In a smart-card operating system, for example, encryption is a minor part of the system and won't be able to use more than half of the available RAM. Obviously, if an algorithm does not fit on the desired CPU, with its particular RAM/ROM configuration, its performance on that CPU family is irrelevant.

For some of the candidates, the performance or RAM requirements can depend on whether encryption or decryption is being performed. It is tempting to consider only one of the two operations, using the argument that the smart card can perform the more efficient side of the operation, and the terminal (with the faster, larger, and more RAM-endowed CPU) can perform the less efficient side. Experience shows that this does not work. Many smart card terminals contain a secure module; in many cases this secure module is itself a smart card chip. In several applications, it is a requirement that two smart cards execute a protocol together, and many existing protocols use both encryption and decryption on the same smart card.

Table 3 compares the RAM requirements for the different AES submissions. This table is identical to that in [SKW+99b], with the exception of MARS.

Algorithm Name	Smart Card RAM (bytes)
MARS	100
RC6	210
Rijndael	52
Serpent	50
Twofish	60

Table 3: AES Candidates' Smart Card RAM Requirements

With its new key schedule "tweak," MARS has on-the-fly subkey generation and appears to require 60 bytes of subkey RAM. When added to the 16 bytes of plaintext and 16 bytes of key, plus other scratch variables, it appears that about 100 bytes of RAM are required. This amount of RAM is much more reasonable than the pre-tweak version, although it is still nearly twice the size of Rijndael, for example.

The new key schedule does allow MARS to fit on small 8-bit CPUs. It should be noted, however, that in this case, the entire rather complex key schedule is recomputed for each block processed, slowing down performance by probably at least a factor of four over a precomputed key schedule

These numbers assume that the key must be stored in RAM, and that the key must still be available after encrypting a single block of text. The results seem to fall into two categories: algorithms that can fit on any smart card (less than 128 bytes of RAM required), and algorithms that can fit on higher-end smart cards (between 128 and 256 bytes of RAM required).[2]

Even if an algorithm fits onto a smart card, it should be noted that the card functionality will include much more than block encryption, and the encryption application will not have the card's entire RAM capacity available to it. So, while an algorithm that requires about 200 bytes of RAM can theoretically fit on a 256-byte smart card, it probably won't be possible to run the smart card application that calls the encryption. For many applications, a RAM requirement of more than 64 bytes just isn't practical.

Given all these considerations, the only algorithms that seem to be suitable for widespread smart card implementation are Rijndael, Serpent, and Twofish.

6 Conclusions

The principal goal guiding the design of any encryption algorithm must be security. If an algorithm can be successfully cryptanalyzed, it is not worth using. In the real world, however, performance and implementation cost are always of concern. For most operational systems, encryption is simply another feature that must be incorporated into a design, and must be traded off with other features. Efficiency, whether software encryption speed, hardware gate count, or hardware key-setup speed, may mean the difference between using AES or a home-grown cipher. Therefore, AES must be efficient.

Efficiency means many different things, depending on context. Efficiency may mean bulk encryption speed in software. Encryption may mean key-setup time in hardware. Efficiency may mean hardware gate count, or speed as a hash function, or speed for short messages on 8-bit smart card CPUs. As a standard, AES must be efficient in all of these meanings, since it will be used in all of these applications.

The most obvious conclusion that can be drawn from this exercise is that it is very difficult to compare cipher designs for efficiency, and even more difficult to design ciphers that are efficient across all platforms and all uses. It's far easier to design a cipher to be efficient on one platform, and then let the other platforms come out as they may. In the previous sections, we have tried to summarize the efficiencies of the AES candidates against a variety of metrics. The next thing to do is would be to assign a numerical score to each metric and each algorithm, then weights to each of the metrics, and finally to calculate an overall score for the different algorithms. While appearing objective, this would be more subjective than we want to be; we leave it as an exercise for the reader.

The performance comparisons will most likely leave NIST in a bit of a quandary. The easiest thing for them to do would be to decide that certain platforms are important and others are unimportant, and to choose an AES candidate that is efficient only on the important platforms. Unfortunately, AES will become a standard. This means AES will have to work in a variety of current and future applications, doing all sorts of different encryption tasks. Specifically:

[2] We do not consider the solution of using EEPROM to store the expanded key, as this is not practical in many smart card protocols that require the use of a session key. All algorithms have significantly reduced RAM requirements if this solution is used.

- AES will have to be able to encrypt bulk data quickly on top-end 32-bit and 64-bit CPUs. The algorithm will be used to encrypt streaming video and audio to the desktop in real time.
- AES will have to be able to fit on small 8-bit CPUs in smart cards. To a first approximation, all DES implementations in the world are on small CPUs with very little RAM. They are in burglar alarms, electricity meters, pay-TV devices, and smart cards. Certainly, some of these applications will use 32-bit CPUs as those get cheaper, but that simply means that there will be another set of even smaller 8-bit applications. These CPUs will not go away; they will only become smaller and more pervasive.
- AES will have to be efficient on the smaller, weaker 32-bit CPUs. Smart cards won't be getting Pentium-class CPUs for a long time. The first 32-bit smart cards will have simple CPUs with a simple instruction set.
- AES will have to be efficient in hardware, in not very many gates. There are lots of encryption applications in dedicated hardware: contactless cards for fare payment, for example.
- AES will have to be key agile. There are many applications where small amounts of text are encrypted with each key, and the key changes frequently. IPsec is an excellent example of this kind of application. This is a very different optimization problem than encrypting a lot of data with a single key.
- AES will have to be able to be parallelized. Sometimes you have a lot of gates in hardware, and raw speed is all you care about.
- AES will have to work on DSPs. Sooner or later, your cell phone will have proper encryption built in. So will your digital camera and your digital video recorder.
- AES will need to work as a hash function. There are many applications where DES is used both for encryption and authentication; there just isn't enough room for a second cryptographic primitive. AES will have to serve these same two roles.

Choosing a single algorithm for all these applications is not easy, but that's what we have to do. And when AES becomes a standard, customers will want their encryption products to be "buzzword compliant." They'll demand it in hardware, in desktop computer software, on smart cards, in electronic-commerce terminals, and in other places we never thought it would be used. Anything chosen as AES has to work in all those applications.

7 Authors' Biases

As authors of the Twofish algorithm, we cannot claim to be unbiased commentators on the AES submissions. However, we have tried to be evenhanded and fair. Many of the performance numbers in this paper are estimates; we simply did not have time to code each submission in optimized assembly language. As more accurate performance numbers appeared, we have updated the tables in this paper. We will continue to do so.

References

ABK98 R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," NIST AES Proposal, Jun 98.

Alm99 K. Almquist, "AES Candidate Performance on the Alpha 21164 Processor," version 2, posted to sci.crypt, 4 Jan 1999.

BGG+99 O. Baudron, H. Gilbert, L. Granboulan, H. Handschuh, A. Joux, P. Nguyen, F. Noilhan, D. Pointcheval, T. Pornin, G. Puopard, J. Stern, and S. Vaudenay, "Report on the AES Candidates," *Second AES Candidate Conference*, 1999.

Bih98 E. Biham, "Design Tradeoffs of the AES Candidates," invited talk presented at ASIACRYPT '98, Beijing, 1998.

Bih99 E. Biham, "A Note Comparing the AES Candidates," revised version, comment submitted to NIST, 1999.

BCD+98 C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, "MARS — A Candidate Cipher for AES," NIST AES Proposal, Jun 98.

DR98 J. Daemen and V. Rijmen, "AES Proposal: Rijndael," NIST AES Proposal, Jun 98.

FKS+00a N. Ferguson, J. Kelsey, B. Schneier, M. Stay, D. Wagner, and D. Whiting, "Improved Cryptanalysis of Rijndael," *Fast Software Encryption, 7th International Workshop,* Springer-Verlag, 2000, to appear.

FKS+00b N. Ferguson, J. Kelsey, B. Schneier, D. Whiting, “A Twofish Retreat: Related-Key Attacks Against Reduced-Round Twofish,” Twofish Technical Report #6, http://www.counterpane.com/twofish-related.html, Feb 00.

Gla98 B. Gladman, “AES Algorithm Efficiency,” http://www.seven77.demon.co.uk/aes.htm, 1 Dec 98.

Gra00 L. Granboulan, “AES: Timings of the Best Known Implementations,” http://www.dmi.ens.fr/~granboul/recherche/AES/timings.html, 15 Jan 00.

KM00 L. Knudsen and W. Meier, “Correlations in RC6,” *Fast Software Encryption, 7th International Workshop,* Springer-Verlag, 2000, to appear.

Knu99 L. Knudsen, “Some Thoughts on the AES Process,” comment submitted to NIST, 15 April 1999.

KS00 J. Kelsey and B. Schneier, “Mars Attacks! Cryptanalyzing Reduced-Round Variants of MARS,” ,” *Third AES Candidate Conference,* 2000, to appear.

KSS00a J. Kelsey, T. Kohno, and B. Schneier, “Amplified Boomerand Attacks Against Reduced-Round MARS and Serpent,” *Fast Software Encryption, 7th International Workshop,* Springer-Verlag, 2000, to appear.

KSS00b T. Kohno, J. Kelsey, and B. Schneier, “Preliminary Cryptanalysis of Reduced-Round Serpent,” *Third AES Candidate Conference,* 2000, to appear.

Lip00 H. Lipmaa, “AES Ciphers: Speed,” http://home.cyber.ee/helger/aes/table.html, 15 Jan 00.

NBD+99 J. Nechvatal, E. Barker, D. Dodson, M. Dworkin, J. Foti, and E. Roback, “Status Report on the First Round of the Development of the Advanced Encryption Standard,” *Journal of Research of the National Institute of Standards and Technology,* v. 104, n. 5, Sept.–Oct. 1999, pp. 435–459.

NIST97a National Institute of Standards and Technology, “Announcing Development of a Federal Information Standard for Advanced Encryption Standard,” *Federal Register,* v. 62, n. 1, 2 Jan 1997, pp. 93–94.

NIST97b National Institute of Standards and Technology, “Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES),” *Federal Register,* v. 62, n. 117, 12 Sep 1997, pp. 48051–48058.

PRB98 P. Preneel, V. Rijmen, and A. Bosselaers, “Principles and Performance of Cryptographic Algorithms,” *Dr. Dobb’s Journal*, v. 23, n. 12, 1998, pp. 126–131.

RRS+98 R. Rivest, M. Robshaw, R. Sidney, and Y.L. Yin, “The RC6 Block Cipher,” NIST AES Proposal, Jun 98.

SKW+98 B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, “Twofish: A 128-Bit Block Cipher,” NIST AES Proposal, Jun 98.

SKW+99a B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, *The Twofish Encryption Algorithm: A 128-bit Block Cipher*, John Wiley & Sons, 1999.

SKW+99b B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, “Performance Comparison of the AES Submissions,” *Second AES Candidate Conference*, 1999.

WS98 D. Whiting and B. Schneier, “Improved Twofish Implementations,” Twofish Technical Report #3, Counterpane Systems, 2 Dec 98.

Efficiency Testing of ANSI C Implementations of Round 2 Candidate Algorithms for the Advanced Encryption Standard

Lawrence E. Bassham III

Computer Security Division
Information Technology Laboratory
National Institute of Standards and Technology

1. Introduction

The evaluation criteria for the Advanced Encryption Standard (AES) Round2 candidate algorithms, as specified in the "Request for Comments" [1], includes computational efficiency, among other criteria. Specifically, the "Call For AES Candidate Algorithms" [2] required both Reference ANSI[1] C code and Optimized ANSI C code, as well as Java™[2] code. Additionally, a "reference" hardware and software platform was specified for testing. NIST performed testing on this reference platform, as well as several others. Candidate algorithms were tested for computational efficiency using the Optimized ANSI C source code provided by the submitters.

This paper describes the testing methodology used in ANSI C efficiency testing, along with observations regarding the resulting measurements. The results of the measurements are included followed by conclusions regarding which algorithms have the most consistent performance across different platforms. Some knowledge regarding compilation and processor architectures is useful in understanding how the data was derived. However, the raw data in the document may be useful without necessarily understanding the derivation.

The testing described in this paper is similar to that done in Round 1. The testing has obviously been restricted to the five Round 2 candidates. Additionally, Timing Tests for the Pentium based platforms has been omitted in favor of Cycle Count testing (see Section 3).

2. Scope

Performance measurements were taken on multiple platforms. These measurements were analyzed to determine the general rankings of the candidate algorithms with respect to one another. NIST is not interested in the absolute value of the performance measurement, but in the relative value of one algorithm's speed when compared with the rest. From an efficiency point of view, NIST does not intend to rank one algorithm as "better" because it is relatively faster

[1] ANSI – American National Standards Institute

[2] Certain commercial products are identified in this paper. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that material identified is necessarily the best for the purpose.

than another algorithm by 0.5%. However, if one algorithm was faster than another algorithm by 50%, then that would be considered a significant difference. NIST is interested in finding the consistent "top performers" on the test platforms by analyzing the performance data for the algorithms and observing natural breaks.

3. Methodology

In the "Call for AES Candidate Algorithms" [2], NIST cited a specific hardware and software platform as the "NIST Analysis Platform" (referred to in this document as the "reference platform") for testing candidate algorithms. This platform consists of an IBM-compatible PC with an Intel® Pentium® Pro™ Processor, 200 MHz-clock speed, 64MB RAM, running Microsoft® Windows® 95, and the ANSI C compiler in the Borland® C++ Development Suite 5.0. Performance measurements were taken on this platform and a large number of additional hardware and software platform combinations. The platforms tested are detailed in Table 1.

Table 1: System Platforms (Hardware/Software) and Compilers Used in Efficiency Testing

Processor/Hardware	Operating System	Compiler
200MHz Pentium Pro Processor, 64MB RAM	Windows95	Borland C++ 5.01 (cycles)
		Visual C++® 6.0 (cycles)
	Linux	GCC 2.8.1 (timing)
450MHz Pentium II Processor, 128 MB RAM	Windows98 4.10.1998	Borland C++ 5.01 (cycles)
		Visual C 6.0 (cycles)
600MHz Pentium III Processor, 128 MB RAM	Windows98 4.10.1998	Borland C++ 5.01 (cycles)
		Visual C 6.0 (cycles)
Sun™: 300MHz UltraSPARC-II™ w/ 2MB Cache, 128 MB RAM	Solaris™ 2.7 (a 64 bit operating system)	GCC 2.8.1
		Sun Workshop Compiler C™ 4.2
Sun: 2*360MHz UltraSPARC-II w/ 4MB Cache, 256 MB RAM	Solaris 2.7	GCC 2.8.1
		Sun Workshop Compiler C 4.2
Silicon Graphics™: 2*300MHz R12000™ w/ 4MB Cache, 512 MB RAM	IRIX64™ 6.5.4 (a 64 bit operating system)	GCC 2.8.1
		MIPSpro C Compiler 7.30

Performance measurements were conducted in two different ways. The first performance test method determines the amount of time required to perform cryptographic operations (e.g., how many bits of data can be encrypted in a second, or how many keys can be setup in a second). This type of test is referred to as a "Timing Test" in this document. The second performance testing method counts the number of clock cycles required to perform cryptographic operations (e.g., how many cycles are consumed in encrypting a block of data, or how many cycles are consumed in setting up a key). This type of test is referred to as a "Cycle Count Test" in this document. The Timing Tests utilized the `clock()` timing mechanism in the ANSI C library to calculate the processor time consumed in the execution of the API call and underlying cryptographic operation under test (i.e., `makeKey(), blockEncrypt(),` and `blockDecrypt()`). The time consumed to perform a particular operation was then used to calculate the bits/second or keys/second speed measure. The Cycle Count Tests counted the actual clock cycles consumed in performing the operation under test (for more information on counting clock cycles see [3]). Because cycle counting utilizes assembly language code in the testing program, interrupts could be turned off during testing[3]. This results in a very accurate measure of the performance of the API calls and the underlying cryptographic operations. Additionally, cycle counting eliminates the variability of the processor speed. The same number of clock cycles are required to perform an operation on a 300 MHz Pentium II processor as on a 450 MHz Pentium II processor; there are simply more clock cycles in a second on a 450 MHz-based system. Cycle counting could only be performed on the Intel processor based systems. This is the only processor used by NIST during Round 2 testing that provides access to a true cycle counting mechanism.

3.1 Cycle Counting Program

For each key size required by [2] (128 bits, 192 bits, and 256 bits) four values are calculated:

- The number of cycles needed to setup a key for encryption;
- The number of cycles needed to encrypt block(s) of data;
- The number of cycles needed to setup a key for decryption; and,
- The number of cycles needed to decrypt block(s) of data.

These values were measured by placing the CPUID and RDTSC assembly language instructions around the NIST API. These instructions were called twice before the cryptographic operation to “flush” the instruction cache (see [3, §3.1]). Additionally, the CLI and STI instructions were used to disable interrupts before testing and enable after testing. This eliminates extraneous interrupts that would skew results. The test program generates 1000 sets of cycle count information as described above for each key size. The values in each category are then sorted, and the median value is determined. A standard deviation is calculated for each test category.

```
makeKey();
cipherInit();
for (r=0; r<1000; r++) {
      cli;              /* Clear Interrupt Flag  */
      cpuid;            /* Clears instruction cache  */
      rdtsc;            /* Read Time Stamp Counter  */
      save counter;
      blockEncrypt();   /*  Perform operation being timed  */
      cpuid;
      rdtsc;            /* Read Time Stamp Counter  */
      subtract counter;
      save counter
      sti;              /* Set Interrupt Flag  */
      }
```

Finally, the average of all values that fall within three standard deviations of the median is determined. This value is the reported average time to perform the specific operation (encrypt, decrypt, or key setup) for a particular key size. Values in this test program are calculated around

[3] Interrupts occur, for example, when the operating system needs to perform some action unrelated to the process that is running. If an interrupt were to occur during cycle count testing, the time spent performing the operating system activity would be included in the time spent on the cryptographic operation. This would lead to inflated and erroneous values for the cycles necessary to perform the cryptographic operation.

the NIST API calls. Results for the Cycle Counting Program can be found in Section 5.1. Pseudo code for the generation of cycle counting information for the `blockEncrypt()` operation is included in Figure 1.

The Cycle Counting Program was run several times with different lengths of data for encryption and decryption to determine if size had any effect on the `blockEncrypt()` and `blockDecrypt()` speeds.

3.2 Timing Program

For each key size required by [2] (128 bits, 192 bits, and 256 bits) four values are calculated:

- The time to setup 10,000 keys for encryption;
- The time to encrypt 8192 blocks of data (8192 blocks*128 bits/block=1048576 bits=1Mbit);
- The time to setup 10,000 keys for decryption; and,
- The time to decrypt 8192 blocks of data (8192 blocks*128 bits/block=1048576 bits=1Mbit).

Analysis of this data was performed in the same way as the cycle count program listed above in Section 3.1 (calculation of standard deviation, median, etc.) Results for the Timing Program can be found in Section 5.2. Pseudo code for the generation of timing information for the `blockEncrypt()` operation is included in Figure 2.

```
makeKey();
cipherInit();
for(r=0; r<1000; r++){
      (Start Timer)
      blockEncrypt(8192 blocks);
      (Stop Timer)
      }
```

Fig. 2: Pseudo code for Time Testing for `blockEncrypt()`

3.2 Compiler Options

PC

On the three PCs used during testing, all algorithms were compiled using the same compiler options. Those options and their effect are:

- Borland:
    - ➢ -Oi Expand common intrinsic functions
    - ➢ –6 Generate Pentium Pro instructions
    - ➢ –v Source level debugging (does not effect speed)
    - ➢ –A Use only ANSI keywords
    - ➢ –a4 Align on 4 bytes
    - ➢ –O2 Generate fastest possible code

- Visual C:
  - /G6 Pentium Pro instructions
  - /Ox Best optimization for speed
- Linux/GCC:
  - -O3 Best optimization for speed

The Borland programs were compiled on the 200 MHz Pentium Pro Reference machine. The Visual C and DJGPP programs were compiled on the 450 MHz Pentium II machine. The Linux operating system was installed on a Jaz drive attached to the 200 MHz Pentium Pro Reference machine. Compilations for GCC under Linux were performed on this machine.

Sun

All algorithms were compiled using the same compiler options. Those options and their effect are:

- GCC: -O3 Best optimization for speed
- Workshop: -xO5 Best optimization for speed

The compilations for the Sun systems were performed on the 300 MHz UltraSPARC II system.

SGI

All algorithms were compiled using the same compiler option. That option and its result is:

- GCC: -O3 Best optimization for speed
- MIPSpro: -O3 Best optimization for speed

The Twofish algorithm compiles on the SGI using the MIPSpro compiler, but results in a Bus Error and a core dump when the `blockEncrypt()` and `blockDecrypt()` functions are invoked. This appears to be a problem with how the compiler is handling byte alignment in the optimized code.

4. Observations

Some of the algorithms use flags to determine which compiler is used. By checking which compiler is used, an algorithm may substitute commands that direct the compiler to insert code to make use of instructions available on the CPU. The most common example of this is the use of the ROTL and ROTR instructions to perform left and right logical rotations, respectively. Using the machine instruction to perform these rotations results in code which is two cycles faster than performing the equivalent sequence of using a pair of shifts and an OR operation. This can provide a performance enhancement on various compilers that other algorithms do not enjoy because they do not perform this type of compiler dependent compilation. The Borland compiler does not make use of the machine instructions of ROTL and ROTR. The Visual C compiler can make use of the machine instructions by using the routines `_rotl()` and `_rotr()` to perform the rotation.

The `blockEncrypt()` and `blockDecrypt()` times improved as the numbers of blocks passed to the algorithm at the same time increased, because the API overhead is averaged over more blocks, and more data is available in the cache. The larger amounts of data are still encrypted and decrypted in ECB mode; however, in operational use, Cipher-Block Chaining (CBC) mode would likely be used. Efficiency testing was not performed in CBC mode because this would add another layer of data processing that has no real impact on the performance of the algorithm, i.e., pre- and post-processing the data before calling the algorithms' internal ciphering routines. In addition, there may be performance characteristics from one algorithm to another, based on whether data is treated as two 64-bit blocks or four 32-bit blocks, but this effect depends on the processor characteristics.

5. Results

5.1 Cycle Count Tables

The values[4] in Ekey, Dkey, Enc, and Dec are all in clock cycles. These values refer to:

- Ekey - The number of cycles needed to setup a 128-bit key for encryption;
- Dkey - The number of cycles needed to setup a 128-bit key for decryption;
- Enc - The number of cycles per block needed to encrypt *n* blocks of data; and,
- Dec - The number of cycles per block needed to decrypt *n* blocks of data.

Note: the data encrypted and decrypted in the cycle count measurements was random (as opposed to using all zero data blocks).

Cycles – Borland C++ 5.01 – 200 MHz Pentium Pro, 64MB RAM, Windows95

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	6815	6814	1097	1049	944	921	937	913	938	914	957	933
MARS-192	7001	7001	1094	1059	947	921	938	913	937	918	956	935
MARS-256	7222	7222	1081	1058	944	926	938	913	939	914	958	932
RC6-128	5171	5170	950	911	630	576	610	556	614	558	629	582
RC6-192	5254	5265	950	914	636	578	609	555	614	558	629	582
RC6-256	5330	5331	949	914	630	576	610	556	614	558	629	582
RIJNDAEL-128	2208	2870	826	836	690	690	685	686	682	681	704	714
RIJNDAEL-192	2972	3786	958	961	823	815	815	808	820	811	850	835
RIJNDAEL-256	3691	4684	1106	1137	982	996	939	946	939	947	961	968
SERPENT-128	12324	12291	3569	3273	3429	3158	3422	3155	3422	3163	3436	3178
SERPENT-192	14389	14398	3574	3301	3429	3159	3420	3147	3424	3165	3438	3176
SERPENT-256	16639	16644	3570	3214	3429	3074	3420	3064	3425	3163	3438	3175
TWOFISH-128	13544	13372	1052	1009	725	681	706	660	708	662	727	687
TWOFISH-192	15707	15544	1052	993	722	675	706	660	708	663	728	686
TWOFISH-256	21344	21181	1049	996	723	679	704	660	708	661	729	682

[4] The relative uncertainty for values in all tables is ≤ 1%.

Cycles – Visual C 6.0 – 200 MHz Pentium Pro, 64MB RAM, Windows95

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	4964	4964	837	754	687	598	681	593	684	595	718	629
MARS-192	4996	4996	821	737	686	601	680	593	683	596	719	629
MARS-256	5185	5185	823	743	689	601	680	593	682	595	720	629
RC6-128	2293	2294	640	627	351	351	340	332	343	334	382	355
RC6-192	2401	2402	640	627	352	351	340	332	343	334	382	355
RC6-256	2512	2513	642	629	352	351	343	332	343	334	382	355
RIJNDAEL-128	1278	1764	1277	1308	1138	1133	1125	1136	1134	1135	1149	1124
RIJNDAEL-192	2002	2566	1512	1574	1368	1362	1358	1365	1361	1372	1388	1365
RIJNDAEL-256	2591	3257	1732	1798	1604	1596	1591	1599	1596	1601	1614	1588
SERPENT-128	7092	7104	1439	1293	1298	1135	1286	1129	1285	1128	1326	1165
SERPENT-192	9048	9035	1455	1294	1295	1135	1285	1126	1285	1126	1326	1168
SERPENT-256	10861	10850	1454	1275	1292	1135	1285	1127	1286	1128	1326	1166
TWOFISH-128	9950	9790	1264	1024	965	725	947	707	950	711	967	740
TWOFISH-192	13298	13136	1265	1020	966	728	947	707	949	721	965	753
TWOFISH-256	18555	18394	1278	1016	965	726	947	707	950	710	966	743

Cycles – Borland C++ 5.01 – 450 MHz Pentium II, 128MB RAM, Windows98

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	6837	6837	1105	1082	947	924	939	913	941	920	986	963
MARS-192	7040	7038	1105	1092	949	919	939	913	937	921	985	961
MARS-256	7249	7249	1105	1082	949	922	936	914	941	921	992	966
RC6-128	5186	5183	984	944	631	578	610	556	617	560	651	598
RC6-192	5279	5279	984	943	631	577	609	555	617	560	651	598
RC6-256	5363	5364	984	944	631	578	609	555	617	560	651	598
RIJNDAEL-128	2254	2912	845	844	689	699	681	692	696	697	777	783
RIJNDAEL-192	2994	3778	983	993	818	814	811	807	826	820	892	896
RIJNDAEL-256	3722	4668	1099	1125	948	958	938	948	954	952	1021	1027
SERPENT-128	11767	11671	3108	2702	2855	2496	2842	2480	2847	2488	2868	2523
SERPENT-192	13872	13852	3108	2705	2856	2478	2842	2465	2847	2467	2868	2505
SERPENT-256	16073	15978	3108	2710	2857	2500	2842	2488	2847	2500	2868	2528
TWOFISH-128	12907	12816	1063	1034	726	677	702	657	708	662	755	708
TWOFISH-192	15311	15219	1061	1031	726	680	704	658	706	665	753	712
TWOFISH-256	20706	20645	1061	1018	727	679	703	657	708	663	754	713

Cycles – Visual C 6.0 - 450 MHz Pentium II, 128MB RAM, Windows98

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	4937	4938	825	734	669	582	658	571	669	583	715	628
MARS-192	4999	4999	825	734	669	578	658	572	667	582	716	629
MARS-256	5175	5175	825	734	668	582	658	572	667	583	716	628
RC6-128	2283	2284	638	622	339	327	321	310	330	320	379	354
RC6-192	2408	2409	638	622	339	327	321	310	330	320	379	354
RC6-256	2519	2520	638	622	339	327	321	310	330	320	379	354
RIJNDAEL-128	1292	1722	987	987	810	801	808	789	826	796	894	866
RIJNDAEL-192	2014	2553	1152	1135	987	969	983	957	1005	972	1079	1039
RIJNDAEL-256	2594	3241	1329	1311	1161	1135	1158	1124	1173	1132	1238	1202
SERPENT-128	6947	6935	1423	1262	1273	1116	1263	1107	1281	1122	1320	1162
SERPENT-192	8857	8857	1423	1280	1274	1117	1263	1107	1281	1122	1320	1162
SERPENT-256	10666	10683	1423	1256	1274	1117	1263	1108	1281	1122	1320	1162
TWOFISH-128	9266	9249	1126	952	802	636	782	615	800	628	831	669
TWOFISH-192	12707	12627	1130	952	802	634	782	616	795	622	832	673
TWOFISH-256	17942	17863	1126	955	802	635	782	616	795	622	832	672

Cycles – Borland C++ 5.01 – 600 MHz Pentium III, 128MB RAM, Windows98

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	6833	6833	1143	1120	951	924	938	913	947	921	976	959
MARS-192	7017	7017	1171	1131	951	926	938	914	940	917	980	959
MARS-256	7245	7245	1143	1120	950	927	939	913	943	918	978	959
RC6-128	5189	5186	1022	982	633	580	610	555	620	567	642	637
RC6-192	5272	5271	1022	982	633	580	610	556	620	567	642	637
RC6-256	5362	5363	1026	982	633	580	609	556	620	567	642	637
RIJNDAEL-128	2213	2862	908	890	692	694	681	681	700	687	757	747
RIJNDAEL-192	2981	3776	1031	1047	820	809	809	799	818	813	883	873
RIJNDAEL-256	3727	4672	1152	1140	959	950	935	937	947	944	1002	996
SERPENT-128	11850	11849	3161	2743	2859	2497	2842	2490	2855	2468	2870	2516
SERPENT-192	13937	13916	3164	2739	2861	2484	2841	2467	2856	2495	2870	2536
SERPENT-256	16133	16114	3165	2737	2859	2500	2841	2485	2849	2483	2869	2536
TWOFISH-128	12938	12861	1085	1057	724	682	704	658	712	667	763	718
TWOFISH-192	15347	15298	1085	1078	727	680	704	659	713	668	764	716
TWOFISH-256	20760	20689	1085	1053	729	681	704	658	718	664	764	713

Cycles – Visual C 6.0 - 600 MHz Pentium III, 128MB RAM, Windows98

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	4934	4936	860	769	668	581	656	569	683	585	708	617
MARS-192	4997	4997	860	769	668	578	656	569	682	585	709	618
MARS-256	5171	5171	860	769	669	581	656	569	682	586	709	617
RC6-128	2278	2279	672	657	339	327	318	307	325	318	366	346
RC6-192	2403	2404	672	657	339	327	319	307	325	318	366	346
RC6-256	2514	2515	672	657	339	327	319	307	325	318	366	346
RIJNDAEL-128	1289	1724	1007	1006	811	802	805	784	824	794	880	848
RIJNDAEL-192	2000	2553	1188	1169	987	966	981	955	1003	971	1069	1023
RIJNDAEL-256	2591	3255	1365	1347	1160	1138	1155	1121	1171	1131	1227	1187
SERPENT-128	6944	6933	1458	1315	1273	1113	1261	1104	1281	1120	1309	1150
SERPENT-192	8853	8853	1459	1297	1273	1116	1260	1102	1281	1123	1309	1151
SERPENT-256	10668	10668	1459	1315	1273	1115	1262	1103	1281	1120	1309	1150
TWOFISH-128	9263	9241	1161	987	802	635	780	613	797	625	828	664
TWOFISH-192	12722	12632	1165	987	802	633	779	613	791	619	828	666
TWOFISH-256	17954	17876	1161	990	802	635	780	613	792	622	828	665

5.2 Timing Tables

Values in the tables are as follow:

- Ekey (time to make a key for encryption) is in Keys/sec;
- Encrypt (time to encrypt) is in Kbits/sec;
- Dkey (time to make a key for decryption) are in Keys/sec; and,
- Decrypt (time to decrypt) is in Kbits/sec.

GCC 2.8.1 - 200 MHz Pentium Pro, 64MB RAM, Linux

	Ekey	Encrypt	Dkey	Decrypt
Mars-128	46729.0	39035.8	46511.6	37135.9
Mars-192	44444.4	39035.8	44642.9	37135.9
Mars-256	42918.5	38855.1	43103.4	37135.9
RC6-128	59523.8	37300.9	58823.5	52454.4
RC6-192	57142.9	37300.9	57803.5	52454.4
RC6-256	56818.2	37300.9	57142.9	52454.4
Rijndael-128	128205.1	42602.6	106383.0	41754.7
Rijndael-192	88495.6	36175.4	74074.1	35562.3
Rijndael-256	74074.1	31551.5	62500.0	30969.4
Serpent-128	16891.9	13052.4	16920.5	16328.2
Serpent-192	13123.4	13052.4	13140.6	16328.2
Serpent-256	10559.7	13052.4	10582.0	16328.2
Twofish-128	14471.8	20671.7	14450.9	22261.8
Twofish-192	11086.5	20671.7	11025.4	22261.8
Twofish-256	8305.6	20671.7	8291.9	22261.8

SGI 300 MHz R12000 w/4MB Cache, 512 MB RAM

	GCC 2.8.1					MIPSpro C Compiler Version 7.30			
	Ekey	Encrypt	Dkey	Decrypt		Ekey	Encrypt	Dkey	Decrypt
Mars-128	60975.6	63581.1	60975.6	66608.8		78125.0	67683.1	78125.0	71124.6
Mars-192	59171.6	63581.1	59523.8	67141.6		76923.1	67683.1	76923.1	70526.9
Mars-256	57803.5	63581.1	57803.5	66608.8		75188.0	67683.1	75188.0	70526.9
RC6-128	147058.8	86522.7	147058.8	98737.7		166666.7	80699.1	166666.7	87424.0
RC6-192	142857.1	86522.7	142857.1	98737.7		161290.3	80699.1	161290.3	87424.0
RC6-256	138888.9	86522.7	138888.9	98737.7		156250.0	80699.1	156250.0	87424.0
Rijndael-128	212766.0	58282.7	161290.3	58282.7		212766.0	74271.7	153846.2	79930.5
Rijndael-192	163934.4	49080.1	125000.0	49368.8		142857.1	63103.0	109890.1	68233.4
Rijndael-256	142857.1	42387.4	108695.7	42819.9		121951.2	54498.1	93457.9	58690.2
Serpent-128	47393.4	42174.4	47393.4	46113.8		57471.3	42819.9	57471.3	45612.5
Serpent-192	37878.8	41963.5	38022.8	46113.8		44247.8	42602.6	44247.8	45612.5
Serpent-256	31250.0	41963.5	31250.0	46113.8		35461.0	42602.6	35461.0	45612.5
Twofish-128	31055.9	59947.9	31055.9	63581.1		41493.8	N/A	41841.0	N/A
Twofish-192	23255.8	60379.2	23310.0	64066.4		32786.9	N/A	33112.6	N/A
Twofish-256	16420.4	59947.9	16447.4	63581.1		22321.4	N/A	22522.5	N/A

Sun 300 MHz UltraSPARC-II w/ 2MB Cache, 128 MB RAM

	GCC 2.95				Sun Workshop Compiler 4.2			
	Ekey	Encrypt	Dkey	Decrypt	Ekey	Encrypt	Dkey	Decrypt
Mars-128	48780.5	29867.3	48543.7	29242.9	52356.0	30081.4	53475.9	29973.9
Mars-192	47393.4	29867.3	46948.4	29141.3	52356.0	30081.4	52083.3	30081.4
Mars-256	46082.9	29867.3	45662.1	29242.9	51020.4	29973.9	51282.1	30081.4
RC6-128	111111.1	20981.8	113636.4	20981.8	111111.1	20470.0	24390.2	20420.2
RC6-192	108695.7	20981.8	108695.7	20981.8	101010.1	20520.1	24449.9	20470.0
RC6-256	105263.2	20981.8	106383.0	20981.8	24390.2	20520.1	98039.2	20470.0
Rijndael-128	172413.8	45612.5	131578.9	38498.6	166666.7	49368.8	117647.1	50864.9
Rijndael-192	140845.1	37805.0	106383.0	32033.2	128205.1	41963.5	85470.1	43261.4
Rijndael-256	117647.1	33042.1	90090.1	27517.1	108695.7	36490.0	73529.4	37467.4
Serpent-128	30120.5	34537.9	30120.5	34969.6	33783.8	32156.0	33898.3	32912.6
Serpent-192	25000.0	34255.9	25000.0	34969.6	27173.9	32033.2	27248.0	32912.6
Serpent-256	21008.4	33841.5	21052.6	34824.5	22421.5	32156.0	22421.5	33042.1
Twofish-128	22321.4	36972.3	22321.4	36020.2	21739.1	41963.5	21739.1	7851.0
Twofish-192	16366.6	36972.3	16366.6	36020.2	16447.4	41754.7	16420.4	7880.5
Twofish-256	11547.3	37300.9	11560.7	36020.2	12285.0	42174.4	12300.1	7865.7

Sun 2*360 MHz UltraSPARC-II w/ 4MB Cache, 256 MB RAM

	GCC 2.95				Sun Workshop Compiler 4.2			
	Ekey	Encrypt	Dkey	Decrypt	Ekey	Encrypt	Dkey	Decrypt
Mars-128	59523.8	36332.1	59523.8	35562.3	65359.5	36649.4	65359.5	36810.1
Mars-192	57803.5	36175.4	57803.5	35412.3	64102.6	36649.4	64102.6	36810.1
Mars-256	56179.8	36175.4	56179.8	35562.3	62500.0	36649.4	62500.0	36810.1
RC6-128	138888.9	26227.2	138888.9	26227.2	142857.1	25587.5	142857.1	25587.5
RC6-192	133333.3	26227.2	135135.1	26227.2	136986.3	25587.5	138888.9	25587.5
RC6-256	129870.1	26227.2	129870.1	26227.2	131578.9	24978.3	131578.9	24978.3
Rijndael-128	217391.3	55215.2	161290.3	47958.3	200000.0	59522.7	142857.1	61260.6
Rijndael-192	172413.8	46886.6	129870.1	39965.3	158730.2	50864.9	107526.9	52454.4
Rijndael-256	142857.1	40940.0	109890.1	34396.3	133333.3	44405.8	88495.6	45612.5
Serpent-128	36101.1	41963.5	36231.9	42819.9	42372.9	39035.8	42372.9	39965.3
Serpent-192	30303.0	41963.5	30303.0	42819.9	34013.6	39035.8	34013.6	39965.3
Serpent-256	25641.0	41963.5	25641.0	42819.9	28328.6	39035.8	28328.6	39965.3
Twofish-128	27322.4	45122.1	27248.0	43039.5	26738.0	53118.4	26738.0	51489.0
Twofish-192	20080.3	44880.8	20080.3	43039.5	20120.7	53456.7	20120.7	51489.0
Twofish-256	14184.4	44880.8	14164.3	43261.4	15015.0	53456.7	15037.6	51806.8

6. Conclusions

6.1 PC

Due to the testing mechanisms used in obtaining data, the most reliable and accurate values obtained for performance measurement of the candidate algorithms are the cycle counting measurements on the PC. Additionally, cycle count values for encryption and decryption were obtained for various data block lengths. These values provide interesting results. For the most part, once the data length was greater than one block (128 bits), the encryption and decryption speeds were consistent within each algorithm. For this reason, NIST focused on the message block length of 128 blocks (2046 bytes), which is a typical size for an electronic mail message. The fastest algorithm for key setup on the PC platform is Rijndael for all compiler and PC hardware/software configurations, followed closely by RC6 and then Mars. Serpent and Twofish are considerably slower than the other algorithms for key setup time. Encryption speed had more variability across compiler and hardware/software platforms. RC6 tends to fall near the top of PC encryption speed followed by Mars, Twofish, and Rijndael. Serpent is consistently at the bottom of the list for encryption speed.

Brian Gladman [4] has performed similar efficiency experiments, the results of which are available on a web page he maintains. The tests that Gladman conducted used code that he developed independently from the submitters' code. Gladman's results are similar to those listed above. Gladman's results for key setup time have the algorithms in basically the same order. The exception being the fact that Serpent's key setup time was greatly improved and ahead of Mars. Again, for encryption speed, Gladman's results coincide with the ordering of the algorithms listed above.

6.2 Sun

The UltraSPARC™ CPU found in the Sun systems on which testing was performed did not allow access to a cycle count mechanism. Performance numbers on these systems are based on the Timing Test Program. Two different compilers were used on the Sun. The data from both these compilers yielded similar results. The fastest algorithms with respect to encryption speed are Rijndael and Twofish, followed by Serpent and Mars, and finally by RC6. However, with respect to key setup Rijndael and RC6 are the fastest followed by Mars which is separated by a wide margin. Serpent and Twofish are last after another wide margin.

Helger Lipmaa reports very similar results on an UltraSPARC-II platform [5]. Lipmaa's table only reports encryption speed. The most noticeable difference is that on his table, the value for the encryption speed of RC6 is closer to those for Mars and Serpent.

6.3 SGI

The SGI system provides another 64-bit processor running the same version of the GCC compiler used for the Sun testing described in Section 6.2. Additionally, the MIPSpro compiler provided another configuration for comparison. The results for these compilers place RC6 as the fastest algorithm for encryption by a wide margin, followed by Mars, Twofish, Rijndael and

Serpent. For key setup, RC6 and Rijndael are the fastest, followed by Mars, Serpent, and Twofish, which are separated by a wide margin.

6.4 Overall Performance

The consistent top performers across all platforms with respect to key setup are Rijndael and RC6. Serpent and Twofish are usually significantly poorer performers; however, Gladman reports a much better value for Serpent key setup, placing Serpent ahead of Mars. Encryption speed values tend to vary much more depending on the platform being analyzed. Rijndael, Mars, and Twofish have the most even encryption performance across platforms – not always the fastest, but never near the bottom of the pack. RC6, on the other hand, was the slowest on the Sun systems but the fastest on the SGI and very nearly the fastest on the PC. Serpent is typically the slowest or towards the bottom of the list on encryption speed across platforms.

7. References

[1] "Request for Comments on the Finalist (Round 2) Candidate Algorithms for the Advanced Encryption Standard (AES)," Federal Register, Volume 64, Number 178, pp. 50058-50061, Sept. 15, 1999.

[2] "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)," Federal Register, Volume 62, Number 177, pp. 48051-48058, Sept. 12, 1997.

[3] "Using the RDTSC Instruction for performance monitoring," http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM, Intel Corporation, 1997.

[4] Brian Gladman, "AES Second Round Implementation Experience," http://www.btinternet.com/~brian.gladman/cryptography_technology/aes2/index.htm, March 1999.

[5] Helger Lipmaa, "AES Ciphers: Speed," http://home.cyber.ee/helger/aes/table.html, 1999.

NIST Performance Analysis of the Final Round Java™ AES Candidates

Jim Dray
Computer Security Division
The National Institute of Standards and Technology
james.dray@nist.gov

March 15, 2000

1. Introduction

NIST solicited candidate algorithms for the Advanced Encryption Standard (AES) in a Federal Register Announcement dated September 12, 1997[1]. Fifteen of the submissions were deemed "complete and proper" as defined in the Announcement, and entered the first round of the AES selection process in August 1998. Since that time, NIST has been working with a worldwide community of cryptographers to evaluate the submissions according to the criteria established in[1]. Five candidates were subsequently chosen to enter the final round of the selection process: MARS, RC6, Rijndael, Serpent, and Twofish.

A previous NIST publication entitled "Report on the NIST Java™ AES Candidate Algorithm Analysis"[2] documents the first round analysis performed by NIST, using the Java Development Kit (JDK) Version 1.1.6. Only IBM has submitted official modifications to their candidate (MARS) prior to the final round. Results of the first round analysis using the JDK1.1.6 are therefore still valid for the other four candidates. The revised version of MARS was tested under both JDK1.1.6 and JDK1.3, to ensure an accurate comparison of the modified algorithm's performance in both environments. Performance data for 128, 192, and 256-bit keysizes are also included in the second round analysis.

The JDK itself has gone through two major revisions since the first round. This paper documents additional performance data for the five AES finalists obtained under JDK1.3, and should be used in combination with the first round NIST Java AES analysis to obtain a complete picture of the characteristics of the finalists in different Java environments. Some background information from the first round analysis is repeated herein for convenience. Comments should be addressed to the author at the email address above.

2. Java Platform

AES candidate algorithm submitters were required to provide optimized implementations of their algorithms in Java and the C language. The rationale for this was to provide more information than could be obtained by testing implementations in a single language, and to take advantage of the hardware independence of the Java virtual machine.

The Java virtual machine presents a uniform abstraction of the underlying hardware platform to a Java application or applet. A Java programmer compiles source code into byte code files, which are then interpreted by the Java virtual machine at runtime (byte code files are also known as class files). In theory, a Java byte code file can be interpreted on any hardware platform running the Java virtual machine without recompilation. Since the virtual machine isolates the Java programmer from the underlying hardware, Java programmers cannot write machine-specific code to take advantage of the unique features of a particular platform. Machine-specific code allows for optimization on a given computing platform, but also eliminates the code portability that is a cornerstone of the Java philosophy.

The Java environment has two characteristics that facilitate the AES evaluation process. First, candidate algorithms written in Java can be easily moved from one platform to another to compare performance on different processors at different system clock speeds. Second, submitters cannot write machine-specific code and so all implementations are on a level playing field.

Java does not provide the level of performance that can be attained in some other languages (C or assembler, for example). However, many applications do not require high-speed encryption of large amounts of data, and cryptoalgorithms implemented in Java are easier to integrate into Java applications. Other languages and hardware implementations will be used for applications where absolute performance is an issue, but there will also be a broad range of applications where the ease of implementing, integrating, and maintaining Java AES code outweighs the performance issue.

3. Evaluation Criteria

The NIST Java AES evaluation process is designed to directly address the criteria published in the Federal Register Announcement[1], Section 4. The goal is to provide objective results that can be clearly quantified for use in the selection process. Sections of the Announcement that describe selection critera relevant to the Java AES analysis are repeated here for convenience:

> ***COST***
>
> ii. *Computational Efficiency:* "…Computational efficiency essentially refers to the speed of the algorithm. NIST's analysis of computational efficiency

will be made using each submission's mathematically optimized implementations on the platform specified under Round 1 Technical Evaluation below."

iii. *Memory Requirements:* "Memory requirements will include such factors as gate counts for hardware implementations, and code size and RAM requirements for software implementations."

ALGORITHM AND IMPLEMENTATION CHARACTERISTICS

i. *Flexibility:*

b. "The algorithm can be implemented securely and efficiently in a wide variety of platforms and applications (e.g. 8-bit processors, ATM networks, voice & satellite communications, HDTV, B-ISDN, etc.)."

ii. *Simplicity:* "A candidate algorithm shall be judged according to relative simplicity of design."

Additionally, in Section 6.B (**Round I Technical Evaluation**):

iii. *Efficiency testing:* "Using the submitted mathematically optimized implementations, NIST intends to perform various computational efficiency tests for the 128-128 key-block combination, including the calculation of the time required to perform:

o Algorithm setup,
o Key setup,
o Key change, and
o Encryption and decryption.

NIST may perform efficiency testing on other platforms."

In condensed form, the published NIST criteria require testing of speed for a set of cryptographic operations, code size and RAM requirements, flexibility, and simplicity of design. Since the candidates have been implemented in Java, flexibility is a given for the reasons discussed in the previous section. The Java AES candidates will run on any device containing a Java virtual machine and adequate memory, although performance will obviously vary depending on the processing power of the underlying hardware.

4. Test Procedures

4.1 Overview

The test results presented here were obtained from the NIST-specified hardware platform and the most recent version of the Java environment available at the time of this writing (JDK1.3, beta release). Results for other hardware/Java virtual machine combinations will be made available on the AES home page at http://www.nist.gov/aes, and in papers submitted to NIST by other organizations[3,4,5]. Detailed test results are presented in tabular form in Appendices A and B, and chart form in Appendix C. All NIST testing was performed through the Applications Programming Interface (API) specified in the NIST/Cryptix Java AES Toolkit. Links to the Toolkit and the Java AES API specification can be found at http://csrc.nist.gov/encryption/aes/earlyaes.htm.

The Java compiler provided with JDK1.3 accepts a command line code optimization switch (-O). However, the JDK1.3 documentation[6] states that this switch “does nothing in the current implementation”. Presumably the compiler accepts the optimization switch for reasons of backward compatibility.

4.2 Procedures

Candidate algorithms were compiled from source files provided by submitters using the JDK1.3 compiler. The resulting bytecode files were packaged into a standard Java ARchive (JAR) file named AESCLASSES.jar.

A Java application was developed to allow testing of any candidate/ keysize/operation combination. The test application instantiates the desired candidate from AESCLASSES.jar, and uses the Java reflection API to invoke the Basic API methods.

500,000 cycles of each candidate/keysize/crypto operation were executed, and the total time was recorded for each combination. Start and stop times were obtained through calls to the System.time.millis() method provided in the Java core library, immediately before and after starting the loop that executed the crypto operations. Charts 1, 2, and 3 present performance data for key setup, encrypt, and decrypt operations, respectively. Data points are included for 128, 192, and 256-bit key sizes. For the majority of candidates, encryption and decryption speed is approximately equal for all three key sizes. Rijndael is a minor exception: encryption speed decreases by approximately three percent for each stepwise increase in key size.

5. Results

In comparison to the JDK1.1.6 performance data presented in NIST’s previous paper[2], the results obtained with JDK1.3 show a striking increase in execution speed for all candidates. On average, the five candidates perform 128-bit key setup operations eleven times faster. The average speed for encrypt and decrypt operations has increased by a factor of five. The same hardware platform and program code (except for MARS) were used for both first round and final round testing, so the overall increase in performance can be attributed to differences in the Java environment. In particular, JIT (Just-In-Time) compilation was not used during the first round performance analysis due to a bug in the JDK1.1.6 JIT implementation that caused problems with certain candidates. Usage of the JIT compiler under JDK1.1.6 increases performance by a factor of ten for most Java programs.

Performance data for the new version of MARS under JDK1.1.6 are presented separately in Appendix A. The test setup for the MARS/JDK1.1.6 analysis was exactly the same as for the other algorithms during the first round, and is described in[2].

In addition to the overall performance increase of the finalists under JDK1.3, there were some changes in the relative ordering of candidates. Most of these changes in order were due to relatively small performance differences, as shown in Appendices B and C. The results for 128-bit keysize operations are summarized below, with candidates ordered from fastest to slowest:

128-bit Key Setup:

JDK1.1.6:	Rijndael, RC6, MARS, Twofish, Serpent
JDK1.3:	RC6, MARS, Rijndael, Serpent, Twofish

128-bit Encrypt:

JDK1.1.6:	RC6, Rijndael, MARS, Serpent, Twofish
JDK1.3:	Rijndael, RC6, MARS, Serpent, Twofish

128-bit Decrypt Operations:

JDK1.1.6:	RC6, Rijndael, MARS, Serpent, Twofish
JDK1.3:	Rijndael, RC6, MARS, Twofish, Serpent

REFERENCES

1. "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)", Federal Register: September 12, 1997 (Volume 62, Number 177), Pages 48051-48058.

2. J. Dray, "Report on the NIST Java™ AES Candidate Algorithm Analysis", http://csrc.nist.gov/encryption/aes/round1/r1-java.pdf , November 8, 1999.

3. A. Folmsbee, "AES Java™ Technology Comparisons", Proceedings of the Second Advanced Encryption Standard Candidate Conference, March 22, 1999, Pages 35-50.

4. K. Aoki, "Java Performance of AES Candidates", Submitted to NIST via email in response to the call for public comments on the AES candidates, April 15, 1999.

5. A. Sterbenz, P. Lipp, "Performance Analysis of Java Implementations of the Second Round AES Candidate Algorithms", Submitted to NIST via email, January 2000.

6. Java™ 2 SDK, Standard Edition Documentation (Version 1.3), http://java.sun.com/products/jdk/1.3/docs/.

APPENDIX A: JDK1.1.6 DATA FOR MARS

Key Size	Key Setup	Encrypt	Decrypt
128 bits	165	462	444
192 bits	244	466	444
256 bits	324	465	445

Table data are presented in kilobits per second.

APPENDIX B: RAW DATA TABLES

Algorithm	setKey128	setKey192	setKey256
RC6	2233	3335	4444
MARS	2110	3131	4131
Rijndael	1191	1574	1733
Serpent	487	734	979
Twofish	286	327	361

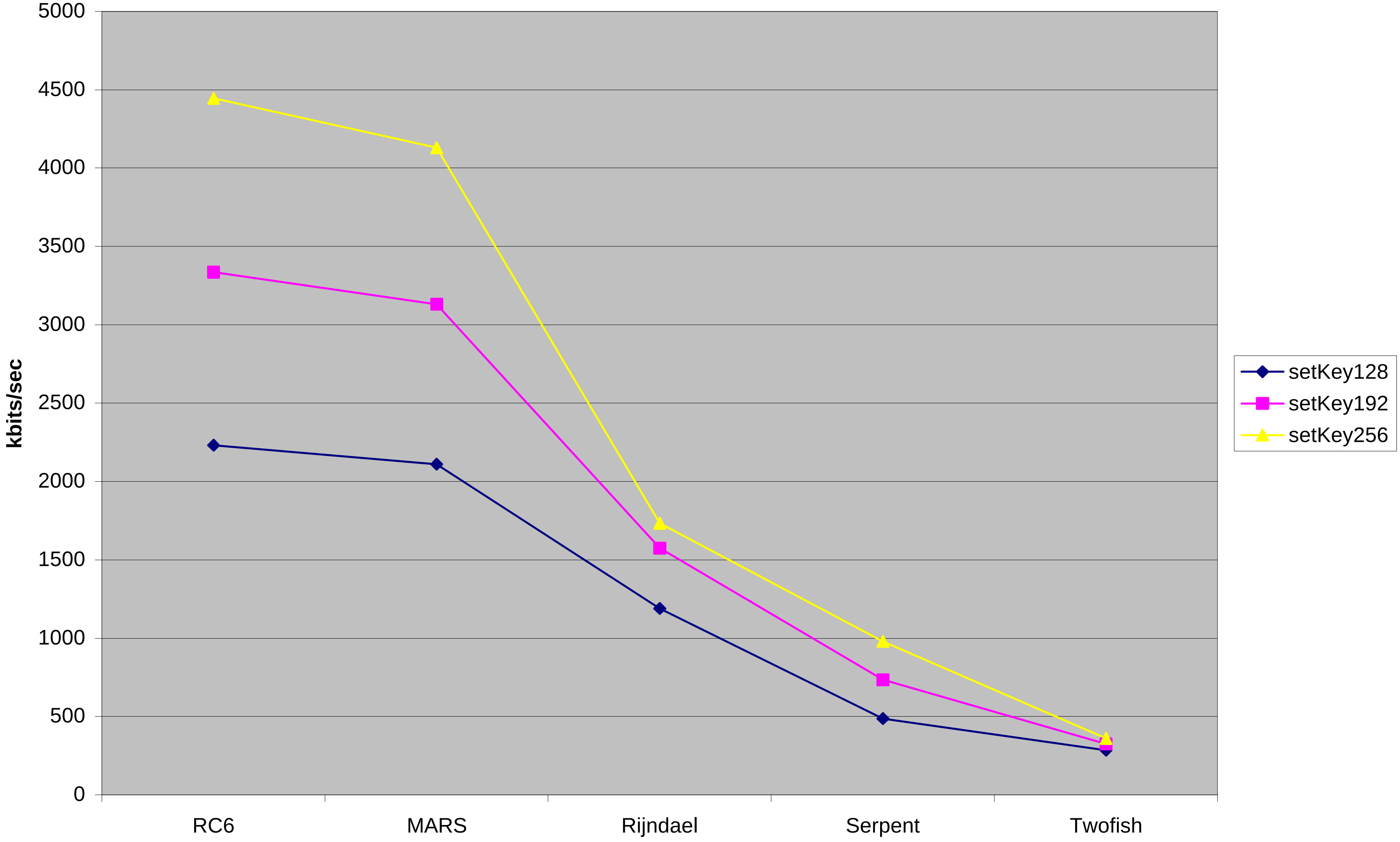
Algorithm	Encrypt128	Encrypt192	Encrypt256
Rijndael	4855	4664	4481
RC6	4698	4740	4733
MARS	3738	3707	3733
Serpent	1843	1855	1861
Twofish	1749	1749	1744

Algorithm	Decrypt128	Decrypt192	Decrypt256
Rijndael	4819	4624	4444
RC6	4733	4698	4740
MARS	3965	3965	3936
Serpent	1873	1897	1896
Twofish	1781	1775	1781

Table data are presented in kilobits per second.

APPENDIX B: PERFORMANCE DATA CHARTS

Chart 1: Key Setup



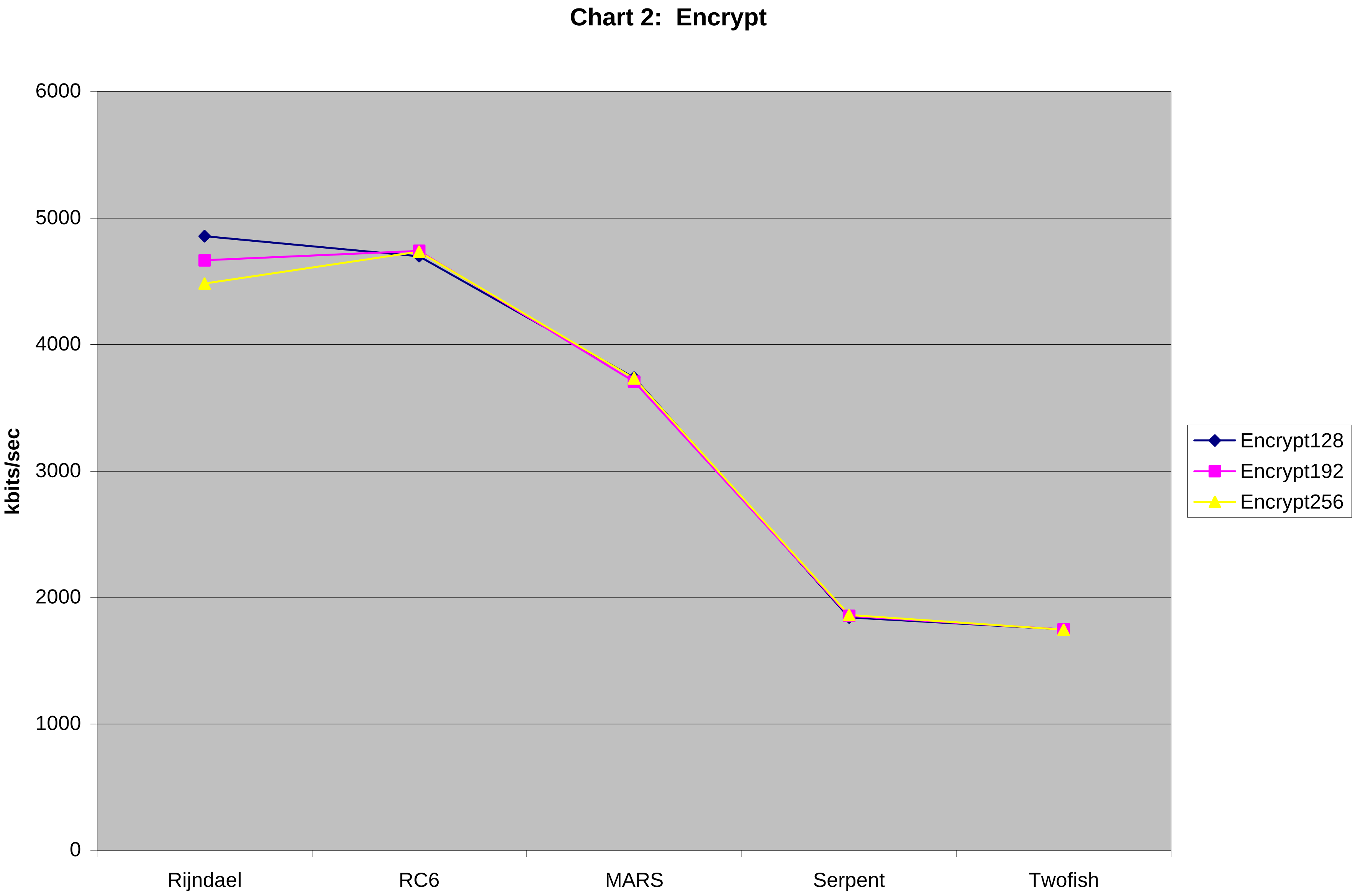
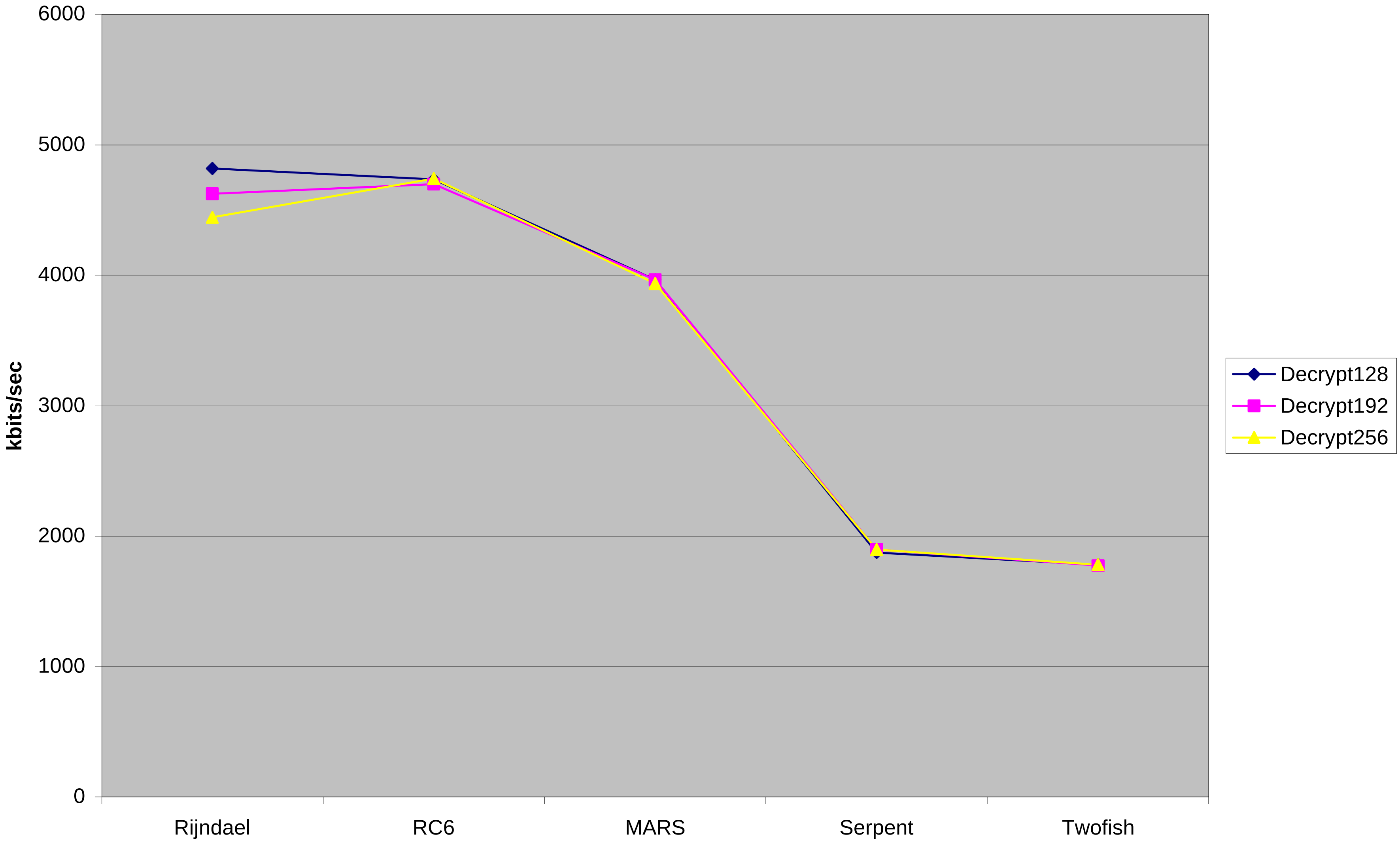


Chart 2: Encrypt
kbits/sec
6000
5000
4000
3000
2000
1000
0
Rijndael
RC6
MARS
Serpent
Twofish
Encrypt128
Encrypt192
Encrypt256

Chart 2.3: Decrypt



Performance of the AES Candidate Algorithms in Java

Andreas Sterbenz, Andreas.Sterbenz@iaik.at
Peter Lipp, Peter.Lipp@iaik.at
Institute for Applied Information Processing and Communications
Graz, University of Technology
Inffeldgasse 16, A-8010 Graz, Austria
http://www.iaik.at

Abstract

We analyze the five remaining AES candidate algorithms MARS, RC6, Rijndael, Serpent, and Twofish as well as DES, Triple DES, and IDEA by examining independently developed Java implementations. We give performance measurement results on several platforms, list the memory requirements, and present a subjective estimate for the implementation difficulty of the algorithms. Our results indicate that all AES ciphers offer reasonable performance in Java, the fastest algorithm being about twice as fast as the slowest.

1. Introduction

The performance of the AES candidates has been the subject of significant discussion, both in the authors' specifications as well as by other parties. Most of this discussion was focused on C and assembler implementations. Some attention has been given to Java implementations but the results were not fully conclusive. This was mostly caused by the fact that the authors' reference Java implementations were evaluated which vary significantly in their coding assumptions and in the degree to which they were subject to optimizations. We intend to fill this gap by evaluating independently developed, consistent Java implementations and comparing the AES candidates' performance to ciphers currently in use.

2. Implementation Notes

The code was developed at the IAIK by Andreas Sterbenz. The AES core code is available under a free license including source at [1] or with a JCE 1.2 compatible API as part of the IAIK JCE library. Serpent S-Box expressions and Rijndael and Twofish setup code are based on C code developed by Dr. Brian Gladman [2].

The design paradigm used is derived from the Java Cryptography Extension (JCE) defined by Javasoft and modified for use within the IAIK JCE library: for each cipher stream a Java object is created which is then initialized with a certain key in either encryption or decryption mode. Then the data to be encrypted is passed to the encrypt (decrypt) method one 128 bit block at a time. Buffering, block chaining, and padding are all performed on a higher level and do not influence the design of the core code. Therefore, for each AES cipher only three methods need to be provided: key setup, encryption, and decryption.

The algorithms have been subject to significant optimization work. The primary focus for the optimization was to maximize encryption and decryption throughput. Secondary and tertiary goals were key setup speed and memory usage, respectively.

3. Java

The Java programming language has become fairly popular in recent years. This is partly due to the fact that Java programs are platform (i.e. processor and operating system) independent in both source and binary form. This is possible by employing a compilation model different from that in most other languages. Instead of compiling source code into machine code for one particular processor family, the compiler produces machine code (called "bytecode") for an imaginary Java Virtual Machine (JVM). At runtime this bytecode is then translated into machine code by a JVM implementation for the particular platform.

This extra step has influences on the programming process when optimizing code. It takes you one step farther away from the hardware making some typical optimization tricks impossible, like for example directly using the processor rotation instruction. Another problem is that a sizable portion of the compilation is delayed until runtime and performed by the JVM. As they are not designed for optimizations this has the effect that those optimizations are not made.

Of course there are several options for the translation of bytecode to machine code. The simplest and most obvious is to use an interpreter: take one JVM instruction at a time and execute the corresponding machine code instruction(s). Much better performance is offered by so-called Just-In-Time (JIT) compilers. They take an entire method and translate it to machine code prior to its first execution, subsequently the generated machine code is executed. JITs are now the common JVM type on most platforms and offer an approximately ten times performance improvement over interpreters. As a third type of JVMs there are hybrid variants aimed at reducing the initial delay caused when the JIT compilers translate a large number of methods at program startup, but this is not relevant for our application.

3.1. Java in Cryptographic Applications

Today the opinion that Java is not the language to be used for cryptographic applications still seems to be popular. Obviously we do not agree. While Java is of course slower than C the difference is typically less than a factor of two, heavily optimized C code excluded, as demonstrated by the results presented in this paper. Although this difference is of course significant Java on today's hardware is faster than C on two year old hardware. The point being that while Java will hardly be the language of choice for high load servers it may well be the choice for medium load servers and especially clients. Add to that handheld and other small devices and performance in Java becomes an issue.

One particular advantage of Java is that there is a well established standard cryptographic API, the JCA and JCE architecture from Javasoft. The success of cryptography libraries in Java including the libraries from the IAIK confirms this position.

4. Evaluation Parameters

The algorithms were implemented in Java. Those implementations were evaluated with respect to three criteria: execution speed, memory usage, and implementation difficulty.

4.1. Execution Speed

For symmetric ciphers there are three components that make up the time required to encrypt some data: static initialization time, key setup time, and data encryption time.

Static initialization is used to perform certain preparation steps, generate constant tables, etc. Because it takes very little time and is largely dependent on the code size vs. speed tradeoff chosen in the implementation it was not measured.

Key setup is used to initialize a cipher for a certain key, i.e. perform round key generation, etc. It is performed once per encryption stream. It may be dependent on whether encryption or decryption mode is chosen and on the key length. For the ciphers analyzed only Rijndael and IDEA have different key setup times for encryption and decryption modes and only Rijndael and Twofish significantly different setup times for different key lengths.

Data encryption time is of course the time it takes to encrypt data bits once the cipher has been properly initialized. The AES candidates are 128 bit block ciphers, that means one encryption operation is performed every 16 data bytes. Again it may vary with the cipher's mode and key length. For all ciphers analyzed the encryption and decryption times are virtually identical and only Rijndael's performance is dependent on the key length.

4.1.1. Key Setup Speed Measurement

Key setup speed was determined as described by the following pseudo code:

```
Repeat 128 times
  Generate 32 random keys
  Start timer
  For each key
    Repeat 1024 times
      Initialize cipher with key
  Stop timer
```

To obtain the final value the average of all measurements within three standard deviations was calculated.

4.1.2. Encryption Speed Measurement

Similarly encryption speed was measured:

```
Repeat 128 times
  Generate a random key
  Initialize cipher with key
  Start timer
  Repeat 2048 times
    Encrypt a 1024 byte array
  Stop timer
```

The same method as above was used to obtain the final value. Note that the same 1024 byte array is encrypted each time which takes full advantage of the CPU caches. In other words, the results presented here are upper boundaries for real world performance.

4.1.3. Environment

The code was compiled using Symantec Visual Cafe 2.5a with optimizations enabled. The results were obtained by running the tests on a machine with an Intel Pentium Pro 200 MHz CPU and 128 MB RAM running Windows NT 4.0 with Service Pack 4. Performance wise this is virtually identical to the NIST reference platform (64 MB RAM and running Windows 95).

However, it should be noted that the actual development and optimization was done on a machine using an AMD K6-2 processor. The optimizing process, which includes trial and error strategies was performed to maximize throughput on this machine and not the reference machine. This may in some cases

	DES	Triple DES	IDEA	MARS	RC6	Rijndael	Serpent	Twofish
Class File Size	n/a	n/a	n/a	9984	1931	4900	12483	5204
Per process memory	5120	5120	0	3220	0	20520	0	6816
Per instance memory	128	384	416	220	432	240	576	4400

Table 1: Class file size and memory usage in bytes.

lead to cases were the performance on the reference machine is not as good as it could be.

4.2. Memory Usage

We give an estimate of the memory required for each of the algorithms. The size of the class file (debugging information removed) is listed to give an idea of the total size, consisting of code size and data like S-Box tables, etc. This is only done for the AES candidates because the other algorithms use a slightly different API which would skew results.

Probably more interesting is the amount of memory required during execution. We list the data memory used obtained by counting the variables used in the source code. Overhead for arrays or data allocated on the stack is not counted as it is fairly small and approximately identical for all of the algorithms.

4.3. Implementation Difficulty

We also assign implementation difficulty "grades" to the algorithms. In difference to the other criteria these were not measured but are subjective estimates for the time it required to arrive at an acceptably fast implementation of the algorithm. If we want to look at it in a quasi formal way we identify the following factors:

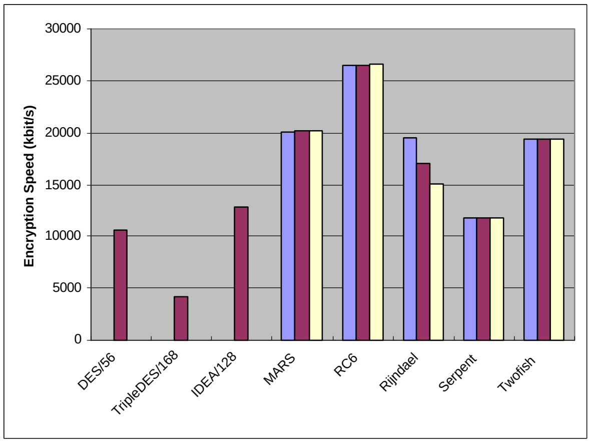
- Time taken to understand the algorithm (at least well enough to be able to implement it).
- Time taken to understand how to efficiently implement the algorithm on a 32 bit platform. As some algorithms need to be coded very differently from their specification in order to be efficient this part may constitute a significant part of the total time.
- Time taken to actually code the implementation.

The first two points are of course to some degree dependent on the documentation provided by the algorithm designers and other parties. Therefore, new or improved documentation may update the results given here.

5. Algorithms

5.1. DES

The Data Encryption Standard (DES) is the current US standard which the AES will eventually replace. It dates back from the 1970s and has become inadequate in particular because of its key length of only 56 bit. DES was designed for hardware implementations and requires tricks to operate moderately fast in 32 bit software implementations. These tricks are not obvious which is why DES only earns a B- for implementation difficulty. However, an advantage of DES over all other algorithms examined except Triple DES is that the encrypt and decrypt operations are identical save for the key schedule resulting in smaller code.



5.2. Triple DES

Triple DES overcomes the limitation of the short DES key length by using three DES cores with separate keys in sequence. This results in an effective strength of 112 bit (meet in the middle attacks) at the price a significant performance drop. Triple DES only performs somewhat faster than one third of the speed of DES (reduced overhead, leaving off the initial and final permutations), which means it is very slow in software. Implementation difficulty is B- as with DES.

5.3. IDEA

IDEA is a 16 bit oriented cipher which uses multiplication modulo 65537 for fast diffusion. Consequently it performs quite well compared to DES (depending on the processors multiplication unit). However, its key setup is quite slow in decryption mode as multiplicative inverses have to be calculated. It has also to be noted that a class of weak keys has been discovered. For implementation difficulty it earns B+ as that is fairly straight forward.

5.4. MARS

MARS is the first of the AES candidates we examine. It uses 8 rounds of unkeyed mixing before and after the core encryption rounds. One of its advantages is that a 32 bit implementation can be written exactly the way the algorithm is specified, also aided by the pseudo code given in the specification. Implementation difficulty B+.

5.5. RC6

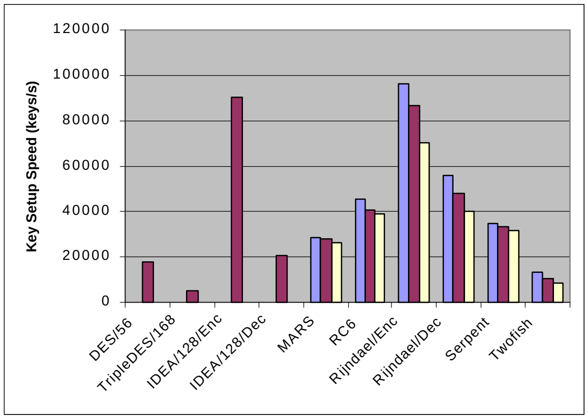
RC6 is a cipher that evolved from RC5. It is very simple to understand and implement and very fast on 32 bit processors; implementation difficulty A. Although the least time was spent on optimizing RC6 it still comes out as the fastest algorithm on almost all platforms.

5.6. Rijndael

Rijndael was designed based on strong mathematical foundations. Implemented on 32 bit processors only table lookup, XOR and shift operations are used. The number of rounds in the Rijndael cipher increases with the key length resulting in decreasing speed for both key setup and encryption. Key setup for Rijndael is very fast for in encryption mode but slower in decryption mode as an additional inversion step is required. Implementation difficulty B.

5.7. Serpent

Serpent was designed for so-called bitslice implementations. The idea is to view a 32 bit register as 32 one bit registers which are operated on by 32 one bit SIMD processors with e.g. logical operations. However, S-Boxes have to be implemented via logical expressions in this mode. Efficient expressions are not trivial to obtain and no expressions are given in the specification, contributing to the B- grade for implementation difficulty. Serpent was designed with a large safety margin of 32 rounds vs. about 20 minimum secure rounds. This results in lower speed, the penalty depending on the JVM implementation and the processor.



5.8. Twofish

Twofish is a very flexible cipher that allows for several implementation options allowing a memory usage vs. key setup speed vs. encryption speed tradeoff. As maximum encryption throughput was desired the "full keying" option was chosen for the implementation. A special property of Twofish is that key dependent S-Boxes are used. This somewhat hurts performance on certain JVMs, in particular when using the Symantec JIT compiler that comes with the JDK on the Windows platform and which was used for the measurements. This means that Twofish may be somewhat faster compared to the other algorithms on other platforms. As Twofish is a quite complicated cipher it earns B- for implementation difficulty.

6. Conclusions

We have analyzed the performance of the AES candidate and other ciphers.

The results for encryption and decryption speed show that RC6 is about 25% faster than the other algorithms. Then MARS, Rijndael, and Twofish follow with virtually identical performance for 128 bit keys, Rijndael being slower for longer keys. Serpent is trailing behind but is still about as fast as IDEA. DES follows with Triple DES far behind. These results are similar to some tests made using C implementations but deviate much from Java studies. The results also show that Java is no more than a factor of 2-3 slower than optimized C code.

The key setup performance is more varied with the fastest AES candidate more than 7 times as fast as the slowest. This appears to be partly due to differing opinions about the purpose of the key schedule. It could be viewed as a one way hash function: accepting an arbitrarily long key, producing output of fixed length (the round keys). All round keys depend on all input bits and obtaining a round key (using some attack) does not yield any information about the original key. Some algorithms try to approximate this ideal while others only generate the necessary key material in a straight forward way.

In any case Rijndael is the fastest algorithm with respect to key setup, although it is not that far ahead when keys longer than 128 bit are used and in decryption mode. Twofish has a fairly slow key setup using this implementation option.

In summary it can be said that if properly implemented all algorithm offer reasonable performance in Java. The results are mostly in line with those obtained by studies evaluating C implementations.

7. References

[1] IAIK. *The IAIK AES home page* http://jcewww.iaik.at/aes/

[2] Brian Gladman. *AES Implementations in C* http://www.btinternet.com/~brian.gladman/cryptography_technology/aes2/aes.r2.algs.zip

[3] X. Lai, J.L. Massey and S. Murphy. *Markov ciphers and differential cryptanalysis* Advances in Cryptology, Proceedings Eurocrypt'91, LNCS 547, D.W. Davies, Ed., Springer-Verlag, 1991, pp. 17-38.

[4] Jim Dray. *Report on the NIST Java AES Candidate Algorithm Analysis* Available from http://csrc.nist.gov/encryption/aes/round2/round2.htm

[5] Lawrence E. Bassham III. *Efficiency Testing of ANSI C Implementations of Round1 Candidate Algorithms for the Advanced Encryption Standard* http://csrc.nist.gov/encryption/aes/round2/round2.htm

[6]

Appendix

This appendix includes the full performance figures as obtained on the reference machine.

Encryption Speed (kbit/s)	**DES (56 bit)**	**Triple DES (168 bit)**	**IDEA**	**MARS**	**RC6**	**Rijndael**	**Serpent**	**Twofish**
128 bit key	10508	4178	12820	19718	26212	19321	11464	19265
192 bit key	n/a	n/a	n/a	19760	26192	16922	11474	19296
256 bit key	n/a	n/a	n/a	19737	26209	14957	11471	19275

Decryption Speed (kbit/s)	**DES (56 bit)**	**Triple DES (168 bit)**	**IDEA**	**MARS**	**RC6**	**Rijndael**	**Serpent**	**Twofish**
128 bit key	10519	4173	13018	19443	24338	18868	11519	18841
192 bit key	n/a	n/a	n/a	19670	24382	16484	11514	18841
256 bit key	n/a	n/a	n/a	19489	24279	14468	11533	18806

Encryption Key Setup (keys/s)	**DES (56 bit)**	**Triple DES (168 bit)**	**IDEA**	**MARS**	**RC6**	**Rijndael**	**Serpent**	**Twofish**
128 bit key	18128	5150	90571	28680	45603	96234	34729	13469
192 bit key	n/a	n/a	n/a	27928	40625	86773	33516	10556
256 bit key	n/a	n/a	n/a	26683	29069	70494	31973	8500

Decryption Key Setup (keys/s)	**DES (56 bit)**	**Triple DES (168 bit)**	**IDEA**	**MARS**	**RC6**	**Rijndael**	**Serpent**	**Twofish**
128 bit key	18039	5136	20737	28743	45709	56017	34687	13469
192 bit key	n/a	n/a	n/a	27917	40625	48324	33560	10550
256 bit key	n/a	n/a	n/a	26731	39028	39963	31973	8531

Session 4:

"Cryptographic Analysis and Properties" (I)

MARS Attacks! Preliminary Cryptanalysis of Reduced-Round MARS Variants

John Kelsey and Bruce Schneier

Counterpane Internet Security, Inc., 3031 Tisch Way, San Jose, CA 95128
{kelsey,schneier}@counterpane.com

Abstract. In this paper, we discuss ways to attack various reduced-round variants of MARS. We consider cryptanalysis of two reduced-round variants of MARS: MARS with the full mixing layers but fewer core rounds, and MARS with each of the four kinds of rounds reduced by the same amount. We develop some new techniques for attacking both of these MARS variants. Our best attacks break MARS with full mixing and five core rounds (21 rounds total), and MARS symmetrically reduced to twelve rounds (3 of each kind of round).

1 Introduction

MARS [BCD+98] is a block cipher submitted by IBM to the AES [NIST97a] [NIST97b], and one of the five finalists for AES. The cipher has an unconventional structure, consisting of a cryptographic "core" in the middle, and a "wrapper" surrounding the core to protect it from various kinds of attack. As with all ciphers, the only way we know to determine the strength of MARS is to try to cryptanalyze various weakened versions of it.

In this paper, we discuss attacks on reduced-round variants of MARS. Because of MARS' unconventional structure, there are many different reduced-round variants worth considering. Here, we focus on two: A variant with the full "wrapper" but fewer rounds of cryptographic core, and a variant with both the core and wrapper reduced by the same number of rounds. In other work [KS00], we have considered the cryptographic core without the wrapper. In view of the stated purpose of the "core" and "wrapper" rounds, we believe the two variants in this paper have a great deal to teach us about the ultimate strength of MARS.

1.1 Current Results

Our results are as follows:

Attacks on Reduced-Round MARS Variants

Reduced-Round Version	**Work**	**Memory**	**Text Requirements**
Full Mixing + 5 Core	2^{232} half encs	2^{236} bytes	8 known plain
Full Mixing + 5 Core	2^{247} partial encs	2^{197} bytes	2^{50} known plain
6 Mixing + 6 Core	2^{197} partial decs	2^{73} bytes	2^{69} chosen plain
0 Mixing + 11 Core	2^{229} partial encs	2^{69} bytes	2^{65} chosen plain

For reasons of both space and clarity of presentation, the attacks against the core rounds only are not included in this paper, and can be found in [KS00].

1.2 Implications of the Results

None of our current results on MARS come close to breaking the full cipher; our best results to date break only 21 out of 32 rounds (and this counts attacking 16 mixing rounds, which are far weaker than the core rounds). However, the attacks in this paper demonstrate ways to attack the MARS structure, and so highlight potential weaknesses of that structure. They also help us to understand how the components of this complex cipher interact to resist attack.

We also introduce a new kind of meet-in-the-middle attack, which may be of independent interest. Although we demonstrate its use initially on MARS, it may be useful against other ciphers, especially other ciphers with heterogenous structures.

1.3 Guide to the Rest of the Paper

The remainder of the paper is arranged as follows: First, we discuss the structure of MARS, and introduce notation and terminology to describe its inner workings. Next, we discuss a set of attacks on MARS with only the number of core rounds reduced. Next, we develop attacks on MARS variants with the same number of each kind of round taken out. We conclude the paper with a discussion of the new techniques we have developed for MARS, the implications of our results, and some open questions.

2 The MARS Structure

The MARS structure can be considered as six different layers through which a plaintext block must pass to become a ciphertext block:

1. Pre-Whitening Layer: The plaintext has 128 bits of key material added to its words modulo 2^{32}.
2. Forward Mixing Layer: Eight rounds of unkeyed mixing operations making extensive use of the MARS S-box.
3. Forward Core Layer: Eight rounds of keyed unbalanced Feistel cipher, using a combination of S-box lookups, multiplications, data-dependent rotations, additions, and XORs to resist cryptanalytic attack.
4. Backward Core Layer: Eight rounds of keyed unbalanced Feistel cipher, using a combination of S-box lookups, multiplications, data-dependent rotations, additions, and XORs to resist cryptanalytic attack.
5. Backward Mixing Layer: Eight rounds of unkeyed mixing operations making extensive use of the MARS S-box.
6. Post-Whitening Layer: The block has 128 bits of key material subtracted from its words modulo 2^{32}.

In this paper, we typically discuss MARS in terms of two different components. The forward and backward core layers together make up the "cryptographic core"; this core looks like a relatively conventional block cipher, and appears to be reasonably resistant to attack. The pre-whitening, forward mixing layer, backward mixing layer, and post-whitening layer together make up the "wrapper," which protects the cryptographic core from various kinds of attack by requiring a large key guess or some clever cryptanalysis to gain access to inputs and outputs of the core. This is a very different block cipher design than is used in the other AES candidates. Among other things, this new design makes it relatively difficult to determine how to come up with reduced-round variants of the cipher to attack.

2.1 The Cryptographic Core

The strength of MARS resides fundamentally in the strength of the core rounds. Both forward and backward core rounds use the same E function, which takes one 32-bit input and two subkey words, and provides three 32-bit words. Each output is combined into one of the three other words. The only difference between forward and backward rounds is the order in which the outputs are combined with the words. The core rounds' strength is based primarily on mixing incompatible operations in the E function, and in their target-heavy Feistel structure, which causes both linear and differential characteristics to quickly spread out into every word. A full description of the MARS core rounds appears in [BCD+98].

The cryptographic core, with a few additional rounds, could stand alone as a cipher; indeed, this would have been a fairly conventional design. Instead, the MARS design team chose to use a smaller number of core rounds,[1] but to surround the core with a "wrapper."

2.2 The Wrapper

The key addition/subtraction and mixing layers surround the core rounds, preventing direct access to the core rounds from either the plaintext or the ciphertext side. While the wrapper itself isn't particularly resistant to cryptanalysis, it is quite different in structure than the core, and it is designed to require guessing of key material before an attacker can learn or control either inputs or outputs to the core.

The mixing layers, like the core, have an unbalanced (target-heavy) Feistel structure, but use only S-boxes and mixing of addition and XOR.

We are a little puzzled by the decision to involve only 128 bits of key material on each side of the core. This leaves the possibility of an attacker guessing his way past either half of the wrapper, and thus seeing either input or output, with a guess of only half the maximum key length. A small change to the design would have involved 256 bits of key material on each side, and thus made partial key

[1] Assuming the same level of performance, adding the wrapper requires reducing the number of core rounds.

guessing worthless as a method of bypassing the wrapper. Below, we consider some attacks that simply guess key material to bypass the wrapper entirely. Inclusion of additional key material would apparently have stopped such attacks at very little cost. While we understand the role of the "wrapper" in helping the core resist attacks, we don't understand why it couldn't fulfill this role just as well with another 128 bits of key material being combined in on each side. Such a change to the design would have rendered many of the attacks we describe in this paper impossible, at a low performance cost.

2.3 Reduced-Round MARS Variants

In a conventional cipher design, the rounds are all more-or-less the same except for subkeys (and sometimes round constants). There is an obvious way to develop weakened versions of such ciphers: simply reduce the number of rounds. Because of the very different roles of the different kinds of rounds in MARS, however, there are a number of reduced-round MARS variants that can teach us valuable lessons about the ultimate strength or weakness of MARS.

Some reduced-round variants we have considered include:

Chopping Off the Beginning or End We evaluate the strength of most ciphers by considering versions with several of the first or last rounds omitted: first the whitening layers and then several rounds of the mixing layers. This isn't a terribly rewarding way to look at MARS, since it omits important parts of the cipher's structure.

Core Rounds Only Because most of the cryptographic strength of MARS apparently resides in the core rounds, it is reasonable to consider the strength of these rounds independently. By developing such attacks, we learn how to attack a fundamental component of the cipher, which may be of use in mounting attacks on the full cipher in the future. For space reasons, most of our analysis of the MARS core is described in another paper [KS00].

Full Cipher with Reduced Core Rounds An alternative way to evaluate the strength of MARS is to consider the full cipher, but with fewer core rounds. This allows us to see how the core rounds might be attacked, even through the whitening and mixing layers that wrap the core rounds of the cipher. It also gives us insights into how strong the core needs to be to allow MARS to resist cryptanalysis.

Symmetric Reductions of the Cipher In the full MARS, there are four different types of rounds, each repeated eight times, for a total of 32 total rounds. It is reasonable to consider symmetric reductions of this; for example, we can consider a MARS variant with only three or four or six of each kind of round. In some sense, this probably provides more information about attacking the full MARS cipher than other kinds of weakened variant, because all the components of the cipher are present.

We believe the last three can teach us many lessons about the ultimate strength of MARS, both in terms of developing tools for attacking the full cipher, and in terms of evaluating how close the best current attacks come to breaking the full MARS.

3 Full Mixing with Reduced Core Rounds

In this section, we consider attacks on a MARS variant with the full “wrapper,” but a reduced “cryptographic core.” These attacks demonstrate how it is possible to mount attacks on a cryptographic core, even through the full wrapper, albeit against a much-weakened core. These attacks penetrate by far the largest number of rounds of the cipher, because they focus on the relatively weak mixing rounds, rather than the much stronger core rounds.

Our attacks in this section are meet-in-the-middle attacks, requiring enormous memory resources to implement, and thus purely academic. In the remainder of this section, we will assume that one memory access to these huge memory devices costs about the same amount of work as a partial encryption. There are ways to trade off time for memory in these attacks, but they generally aren’t useful in the context of these attacks.

3.1 A Straightforward Meet-in-the-Middle Attack on Five Core Rounds

Consider MARS with full mixing and whitening layers, but with the core reduced to three forward and two backward core rounds. This is vulnerable to a meet-in-the-middle attack as follows:

1. Request eight plaintext/ciphertext pairs.
2. From the plaintext side, guess:
   (a) The 128-bit pre-whitening key.
   (b) The 62-bit first round key.
   (c) $K^{\times}$ and the low nine bits of K^{+} for the second round.
   (d) This yields knowledge of $A_2 = D_3 >>> 13$. Compute this value for all eight plaintexts, and put the resulting 256-bit value in a sorted list.
3. From the ciphertext side, guess:
   (a) The 128-bit post-whitening key.
   (b) The 62-bit last round key.
   (c) $K^{\times}$ and the low nine bits of K^{+} for the next-to-last round.
   (d) This yields knowledge of $A_2 = D_3 >>> 13$. Compute this value for all eight ciphertexts, and search the sorted list from the plaintext guesses for a match on this 256-bit value.

This attack passes through 16 mixing rounds and 5 core rounds (thus 21 rounds total), at a cost of about 2^{232} half encryptions’ work (that is, 2^{229} work for each of the eight texts), and about 2^{236} bytes of memory. The memory requirements are totally unreasonable in practice, so this attack is purely academic.

Summary of Results

Attack On:	Full Mixing Plus Five Core Rounds (21 total rounds)
Attack Type:	Meet-in-the-Middle
Work:	2^{232} half-encryptions
Memory:	2^{236} bytes
Texts:	Eight known plaintexts

3.2 The Differential Meet-in-the-Middle Attack

Here, we introduce the concept of a differential meet-in-the-middle attack. This attack is related to the attack on the Mansour-Even construction by Daemen [Dae95], the attack on DESX by Kilian and Rogaway [KR96], and the inside-out attack of Wagner [Wag99].

In a standard meet-in-the-middle attack, we guess some key from the first and second halves of the cipher, and then match on some middle value. For example, an attack on double-DES starts by getting two plaintexts and their corresponding ciphertexts. We then guess the key for the first DES encryption, and for each such key guess, we compute the middle value from the two plaintexts if they were encrypted under that key. This is stored in a sorted list. We then guess the second DES key, and compute, for each guess, the middle value from decrypting the two ciphertexts. These values are searched for in the sorted list. When we find a matching value, it is very likely that this corresponds to the right key.

This attack can be generalized. For example, it is not necessary that the whole intermediate value to an encryption be computed; we can compute a single bit from each direction, and then examine more plaintext/ciphertext pairs. Similarly, if we can compute some checksum from intermediate values we reach by key guesses from the plaintext and ciphertext sides, then we need never have any knowledge of actual intermediate text values, as in [KSW99].

An extension to this idea allows the use of probability one differentials through some intermediate part of the cipher. Consider the truncated differential $(0, 0, 0, \delta_0) \rightarrow (\delta_1, 0, 0, 0)$, which goes through three MARS core rounds with probability one. The truncated differential works the same way in reverse, naturally. This means that if we see a right input pair (a pair with difference $(0, 0, 0, \delta_0)$), we will also see a right output pair (a pair with difference $(\delta_1, 0, 0, 0)$).

In a meet-in-the-middle attack, we must find some value that can be computed from both the top (input) and the bottom (output) of the cipher with a key guess, build a sorted list of these values, and look for pairs of keys that match on these values.

With these differentials, we can compute such a value as follows:

1. Get about 2^{50} known plaintexts and their corresponding ciphertexts. Label each plaintext/ciphertext pair with an index number, $0..2^{50} - 1$.
2. Guess part of the key from the top, and compute intermediate states for each plaintext given that key guess.
3. Sort the plaintext-intermediate values on their first three words.
4. Go through these values, and note each pair of texts that matches on their first three words by their index numbers. List these in sorted order, lower index number first in each pair. We expect about eight of these pairs.
5. Guess part of the key from the bottom, and compute intermediate states for each ciphertext from that key guess. Sort the ciphertext-intermediate values on their last three words.

6. Go through these values, and note each pair of texts that matches on their last three words by their index numbers. List these in sorted order, lower index number first in each pair. We expect about eight of these pairs.
7. Because the differential has probability one in both directions, there must be the same number of these pairs, and the pairs must be identical, from both plaintext and ciphertext. All we've done here is to list which pairs have the right input and output XOR differences to fit this truncated differential.

From this, we now have a "checksum" (the right pair indices) that we can compute across three MARS core rounds. (We can easily restrict the checksum's size to four or eight matching pairs. The indices of the pairs must be put in some standard order; for example, note each right input pair of indices in sorted order, and then sort the pairs in order of each pair's lowest index number.) This checksum costs about $50 \times 2^{50} \approx 2^{56}$ work to find for any block of 2^{50} texts. We can thus do the following differential meet-in-the-middle attack on the full MARS mixing layers plus five rounds of core:

1. Get 2^{50} known plaintext/ciphertext pairs, and label each by an index number as described above.
2. From the plaintext side, guess the pre-addition key and the first core round key, a total of 2^{190} different key guesses.
3. For each key guess, take the predicted inputs to the second core round, and compute the input right pair indices as described above. This takes about 2^{56} work per key guess. Write the input right pair indices to a huge list, one entry per key guess with the first eight right input pair indices in sorted order.
4. Do the same thing from the bottom, continuing to add entries to the huge list.
5. Sort the huge list, which will now have 2^{191} entries in it, and should thus take about $191 \times 2^{191} \approx 2^{199}$ work to sort.
6. Find the match between key guesses from the plaintext and ciphertext sides.

The total work done is thus $2^{190} \times 2^{56} \times 2 + 2^{199} \approx 2^{247}$. The attack recovers all key material used in the pre- and post-addition/subtraction keys, and the first and last core rounds' values, as well. The total memory taken is 56×2^{191} bytes.

Summary of Results

Attack On:	Full Mixing Plus Five Core Rounds (21 total rounds)
Attack Type:	Differential Meet-in-the-Middle
Work:	2^{247} partial encryptions
Memory:	2^{197} bytes
Texts:	2^{50} known plaintexts

3.3 Tradeoffs Between Differential and Conventional Meet-in-the-Middle Attacks

Note that the differential meet-in-the-middle attack requires slightly more work but considerably less memory than the conventional meet-in-the-middle attack.

The advantage of the differential meet-in-the-middle attack is that it allows us to pass through three core rounds for free; the disadvantage is in the cost of detecting the property that passes through those three core rounds for free, and the far larger number of known plaintexts required. This tradeoff determines which attack is best-suited for a given cipher and attack model.

For reference, we will point out that the differential meet-in-the-middle attacks can be used with less memory against smaller numbers of core rounds. For example, we can use the same truncated differential and filtering process against three rounds of core, dropping the total memory required to about 2^{133} bytes of memory, at a work factor of about 2^{185} partial encryptions.

We have considered ways of extending the differential meet-in-the-middle attack another round. Unfortunately, there are complications involved in using either differentials with probability substantially lower than one, or in using differentials that don't run both directions with approximately equal probability.

3.4 Using Lower Probability Differentials

Consider a differential with probability 1/2 through several rounds of some cipher, and assume we must find four input right pairs that are also output right pairs. The problem is that we must have an exact match for the final sorting and searching phase of the meet-in-the-middle attack to work. The only way we can see to mount the attack in this situation is to generate and store many different input right pairs, in hopes that one will consist of all successful differentials, and thus, right output pairs.

The most efficient way to do this will probably be to find R right input pairs, and add to the sorted list of input right pairs every possible 4-tuple of the pairs, and then to do the same with the right output pairs. That will involve R choose 4 entries in the list, and we can expect it to work if we expect at least 4 of the R input right pairs to result in output right pairs. The number of expected right output pairs from R right input pairs is binomially distributed; for reference, with nine right input pairs, we expect four right output pairs with probability 1/2. With twenty right input pairs, we have about a 0.94 probability of getting some subset of four right input pairs.

This implies an unpleasant tradeoff between probability of the differential used, and memory required for the attack. Consider the following numbers, which describe the impact of using lower-probability differentials on the difficulty of the attack. These numbers are for parameters that give the attack an approximate probability of success of 1/2.

Memory vs. Probability Tradeoff

Prob. of Characteristic	Num. Right Input Pairs Required	Num. Entries in Sorted List per Key Guess
0.9	5	5
0.5	9	126
0.1	47	178365
0.01	467	1.96×10^9
0.001	≈ 5000	2.60×10^{13}

As a rule, multiplying the number of entries in the sorted list per key guess by N multiplies the size of that list by N, which multiplies the work involved in handling it by $N \log N$. We thus have great difficulty in using differentials with very low probabilities.

Truncated Differentials with Substantially Lower Probabilities in the Decryption Direction Normal differentials must have the same probability in both directions. (This can be established by a simple counting argument.) However, truncated differentials, which don't specify the whole difference, can have different probabilities in different directions. For example, in the MARS forward core rounds, the following four-round differential has probability one:

$$(0, 0, 0, 2^{31}) \rightarrow (?, ?, (\text{low 12 bits} = 0x1000), 2^{12})$$

However, this truncated differential cannot be run backwards with reasonable probability. There are about $2^{211} + 2^{127}$ pairs of inputs that will yield this output difference; of these pairs, only about 2^{-84} have input difference $(0, 0, 0, 2^{31})$. This makes the attack much more costly; in fact, our best methods to mount the attack in this case allow us to attack the full mixing and four rounds of core, but not five rounds of core.

3.5 Boomerang Meet-in-the-Middle Attacks

We have also considered using the same kind of technique, but with boomerangs [Wag99] (basically, 4-tuples with a differential relationship between all four texts in the middle of the cipher) instead of individual ciphertexts. The problem of detecting when we have the expected boomerangs is difficult; thus far, we have been unable to find a way to do this that isn't far costlier than the rest of the attack can afford.

3.6 Other Techniques

In this section, we have focused on meet-in-the-middle attacks, because these are the most obvious kinds of attacks to consider. However, there are other attacks that might be useful against this kind of reduced-round MARS version. For example, we might guess our way past the pre-addition key and forward mixing layers, and look for a set of text pairs whose properties will show through eight rounds of backward mixing layer. We haven't yet found an effective way to do this for all eight backward mixing rounds, but research is ongoing.

4 Symmetric Reductions of the Cipher

MARS consists of eight rounds each of four kinds of round functions. A natural way to derive a reduced-round version of MARS to analyze is to consider k rounds of each kind. For example, when $k = 2$, we have eight total rounds; when

$k = 3$, twelve; and when $k = 4$, sixteen. Cryptanalysis of such reduced-round versions of the cipher allows us to learn important lessons about how to attack a the general MARS structure.

Our attacks typically work as follows:

1. First, we choose N batches of input pairs, so that one such batch is likely to consist of many pairs that have some differential after the mixing layer.
2. We then exploit some differential property that passes through the core rounds, leaving a detectable differential property somewhere near the end of the cipher.
3. Finally, we guess enough key material at the end to detect the detectable property; the partial key guess that allows us to detect the differential property is the correct one.

4.1 Attacking MARS Symmetrically Reduced to Eight Rounds

When $k = 2$ (eight rounds total), we have a cipher that is obviously not very strong. It is still worthwhile to consider how this might be attacked, in part to help develop techniques for attacking stronger versions. Recall that this cipher consists of the key addition, the first two forward mixing rounds, two forward core rounds, two backward core rounds, the last two backward mixing rounds, and the key subtraction.

For this version, we can simply use one of the meet-in-the-middle attacks discussed in the previous section, since there are only four rounds. However, we can do much better than that.

Our attack works as follows:

1. We choose N batches of eight pairs each, where $N \leq 2^{40}$. The batches will be described below; one batch will have all eight pairs with difference $(0, 0, 0, 2^{31})$ after the forward mixing layer.
2. In the right batch, this passes through the four core rounds with probability one, leaving $(?, ?, ?, 2^{12})$.
3. We guess a few bits of subtraction key at the end of the cipher, and thus distinguish the right batch from all the wrong batches. If we guess m bits of effective key, we will need about 2^{m+44} partial decryptions to distinguish the right batch from the wrong batches.

Choosing the Batches The first step to this attack is to get pairs of texts through two forward mixing rounds, so that we have pairs with the difference $(0, 0, 0, 2^{31})$ in the input to the first core round.

Our plan is to put a 2^7 difference in A, and an offsetting difference T in B, and finally a difference to cancel A's difference in D. To simplify the filtering problem at the end, we will choose *batches* of eight pairs of texts, so that one of the batches will give us eight right pairs through the forward mixing layer.

Choosing A, A^* We show the first difference as being 2^7, in A, which passes through the key addition with approximate probability 1/2, and then generates expected difference T in the output to the first use of S-box s_0 with probability 2^{-7}. This then has probability of 2^{-8}. This is based on simply looking for a pair of S-box inputs, $(u, u \oplus 2^7)$, such that $s_0[u] \oplus s_0[u \oplus 2^7] = T$. We look at all 128 such pairs, and use the difference with the lowest weight in its low 31 bits, for reasons that will become clear momentarily.

For each batch, we hold the low eight bits of A constant. For one such value, $A, A^* = A \oplus 2^7$ will leave a 2^7 difference in A, and a T difference in the output from the first s_0.

Choosing B, B^* The second difference is shown as being T, in B. This passes through the key addition with approximate probability $2^{-w(T)}$, where $w(T)$ is the Hamming weight of the low 31 bits of T. If we get T as the XOR differences in both line B and the s_0 output from A, they cancel out with probability one. (We can also consider a mod 2^{32} difference T that passes through the key addition with probability one, and cancels out the T additive difference in the s_0 output with probability about 2^{-w}; naturally, there is no difference in the probabilities involved.) Recall that we chose $u, u \oplus 2^7$ in A to minimize $w(T)$. Let us assume that the minimum value for $w(T)$ is 12. Then, we have about 2^{-12} probability of finding a pair B, B^* such that their difference after the key addition is T, simply by using the rule that $B^* = B \oplus T$. We can actually do somewhat better than this in our selection of batches.

Building the Batches The third difference is shown as being 2^{31}, in D. This difference passes through all XORs and additions with probability one.

We can thus build batches of $(A, B, C, D), (A^*, B^*, C, D^*)$ pairs. Each batch of eight pairs contains the same low-order eight bits for A and all the same bits for B. There are thus $2^{24} \times 2^{32} \times 2^{32} = 2^{88}$ possible pairs for each batch, and there are 2^{40} batches possible, and about $2^{12} \times 2^8 = 2^{20}$ expected to be necessary.

We build 2^{20} batches of eight pairs, for a total of 2^{24} chosen plaintexts.

Guessing Key at the End After the core rounds, input pairs with difference $(0, 0, 0, 2^{31})$ must have output difference $(?, ?, ?, 2^{12})$. We must thus learn the value of D in the output from the core rounds. To do this, we must guess about 12 bits of the subtractive key for C, and all of the subtractive key for D. Using these guesses, we can derive the values for D after the core rounds. We must do this for all 2^{24} texts. We thus have total work of about $2^{24} \times 2^{44} = 2^{68}$ partial decryptions. (We suspect that there are better attacks for $k = 2$, but that these attacks don't generalize for larger k values.)

Summary of Results

Attack On:	MARS Symmetrically Reduced to Eight Rounds
Attack Type:	Differential
Work:	2^{68} partial decryptions
Memory:	2^{29} bytes
Texts:	2^{25} chosen plaintexts

4.2 Extending the Attack to $k = 3$ (Twelve Rounds)

We now consider an attack on MARS symmetrically reduced to 12 rounds. Again, the cipher is obviously not very strong. However, the structure is beginning to add difficulties to the attack. We use a boomerang-amplifier to cover the six core rounds with probability 2^{-96} for each pair of pairs with difference $(0, 0, 0, 2^{31})$ into the core rounds. With about 2^{50} right pairs into the core rounds' input, we expect to see about four right pairs of pairs. These pairs will then pass through six more rounds, and can be detected by examining the whole output blocks from all the texts.

Requesting Inputs We use the same input structure as before, but instead of requesting eight pairs for each batch, we request 2^{50} pairs for each batch.

The Boomerang Amplifier The boomerang-amplifier attack is introduced in [KS00], and is based on the concept of "boomerangs," as described in [Wag99]. The basic idea of the attack involves a property occurring in pairs of pairs of texts.

For the MARS core, we use the batches of input pairs described above to try to find a batch of 2^{50} pairs of texts, all of which will have difference $(0, 0, 0, 2^{31})$ in the input to the first core round. Since there is a probability one differential for the core rounds, $(0, 0, 0, 2^{31}) \rightarrow (2^{31}, 0, 0, 0)$, this means that all 2^{50} pairs of the batch will have difference $(2^{31}, 0, 0, 0)$ after three core rounds.

Consider the set of 2^{50} of these pairs in the batch. We will refer to these pairs as $((W_0, W_0^*), (W_1, W_1^*), ..., (W_i, W_i^*))$ in input to the core rounds, and as $((X_0, X_0^*), (X_1, X_1^*), ..., (X_i, X_i^*))$ after round three. There are about 2^{99} *pairs of pairs*. That is, there are about 2^{99} different ways to choose two of these pairs out of this batch and look at them together; for example, $((X_i, X_i^*), (X_j, X_j^*))$. Now, consider the difference $(0, 0, 0, a)$, where a is unknown. For any pair of texts to have such a difference, they must collide in 96 bits; the difference thus is expected to occur in 2^{-96} of all random pairs of texts. Thus, if X_i, X_j can be considered as a more-or-less random pair of texts (and they apparently can), then the probability that each i, j pair will have this difference is 2^{-96}, and since we have 2^{99} such pairs, we expect about eight pairs X_i, X_j with this difference. However, we know that $X_i \oplus X_i^* = (2^{31}, 0, 0, 0)$ for all i. This lets us algebraically show that when $X_i \oplus X_j = (0, 0, 0, a)$, $X_i^* \oplus X_j^*$ must also equal $(0, 0, 0, a)$. This boomerang structure thus amplifies the effect of the low-probability event, making it detectable, since when this happens we get *two* pairs of texts that follow the truncated differential $(0, 0, 0, a) \rightarrow (b, 0, 0, 0)$ over three rounds.

Distinguishing the Right Key Guess To distinguish the right key guess, we examine the result of trial partial decryption of a whole batch of pairs at a time. Let Y_i, Y_i^* be the results of encrypting input pair W_i, W_i^* through the whole cryptographic core in some batch. We build a list of all the Y_i and Y_i^* values. We then sort this list on its low-order 96 bits. Next, we go through the list, and for each pair Y_i, Y_j or Y_i, Y_j^* that matches in those last 96 bits, the pair i, j is added to a sorted list of pairs that collided. Finally, we *count* the number of times each i, j appears in the list. When we see two or more instances of the same i, j occurring twice, we are extremely likely to have a correct key guess.

Recall that we expect eight pairs i, j such that $X_i \oplus X_j = X_i^* \oplus X_j^* = (0, 0, 0, a)$ These will inevitably lead to eight pairs i, j such that $Y_i \oplus Y_j = (b, 0, 0, 0)$ and $Y_i^* \oplus Y_j^* = (b', 0, 0, 0)$. (The property works just as well if the collision occurs between X_i and X_j^*, naturally.)

The probability of any given i, j pair having this property after a random permutation has been applied to it is 2^{-192} Since there are 2^{99} pairs in each batch, we expect no such i, j pairs. The probability of seeing two such pairs in a batch (that is, among 2^{99} potential pairs) is about 2^{-186}. We will be examining 2^{20} different batches, each under 2^{128} different keys, so we'll have 2^{148} total batches to examine in this way. So with overwhelming probability, there will be only one partial key guess that will give us two or more such i, j pairs.

Summary of the Attack The attack on 12 rounds ($k = 3$) makes use of a boomerang amplifier. It requires about $2^{20} \times 2^{48} \times 2 = 2^{69}$ texts, about 2^{25} bytes of random-access memory (to hold a batch of texts at a time), and about 2^{73} bytes of sequential memory to store all the ciphertexts so we can apply our guesses to them. The attack also requires about $2^{128} \times 2^{69} = 2^{197}$ partial decryptions, each consisting of about one quarter of the cipher.

Summary of Results

Attack On:	MARS Symmetrically Reduced to 12 Rounds
Attack Type:	Boomerang Amplifier
Work:	2^{197} partial decryptions
Memory:	2^{73} bytes
Texts:	2^{69} chosen plaintexts

5 Conclusions

In this paper, we have developed several new attacks on reduced-round versions of MARS. While none of these attacks is able to break the full cipher, we feel that these results provide valuable insights into the security of MARS. We regard these results as preliminary, and would be unsurprised to see moderate improvements in any of our attacks. However, if major improvements in the results are possible, we expect that they will require new techniques. Below, we describe some ideas for additional attacks on reduced-round MARS variants.

5.1 Lessons from the Analysis

The results in this paper show the overwhelming importance of the strength of the MARS cryptographic core; we can attack the full mixing layers with only five core rounds, a total of 21 rounds, but can currently attack no more than 11 core rounds.

Our results also show how the "wrapper" layers protect the core rounds from attacks that require large numbers of chosen plaintexts or chosen ciphertexts.

Finally, our results demonstrate that, when evaluating a fundamentally new cipher design, it is important to be able to innovate—to develop new techniques to attack the cipher, rather than merely reusing the standard differential and linear attacks. Because MARS is such an unconventional block cipher, we needed to develop new attacks to get very far in our analysis.

5.2 Why This Is Important

The only way we know of actually determining the strength of a cipher is to try to attack it, including reduced-round versions. Proofs of security have proven unreliable; security arguments based on estimates of the best differential and linear characteristics tell us little about what other attacks may be done; design principles that protect against some attacks sometimes allow new attacks in their place; as in Square, where the use of the MDS matrix made differential attacks extremely difficult, while allowing Knudsen's dedicated attack. The history of cryptography is littered with ciphers whose designers were convinced of their security, but whose attackers were not. Without a solid understanding of the security of each of the AES finalists, NIST and the cryptographic community will likely make a final decision on AES based only on performance.

In this paper, we have done some very preliminary analysis of two versions of MARS with reduced rounds. MARS is a complicated enough design that beginning to analyze it involves a significant investment of time (though even conceptually very simple ciphers seem to have much the same property). We hope to see our work spur others to go beyond the very preliminary results in this paper.

5.3 Ideas for Future Attacks

We have spent considerable time trying to get boomerangs to work within meet-in-the-middle attacks. A "boomerang-in-the-middle" attack would go through six rounds for free, and thus would be quite powerful. Similarly, there is a seven-round impossible differential through the core rounds; we are as yet unable to find a way to use either of these ideas in a meet-in-the-middle attack. The underlying problem in the case of the boomerang-in-the-middle attack is that a boomerang 4-tuple can be identified only by considering both input and output simultaneously. We have not been able to find a way around this problem so far. The underlying problem with the impossible differential meet-in-the-middle attack is that we can rule out candidate key guesses only by (again) examining right pairs for both input and output simultaneously. We are still looking for a way around this problem, or for a proof than none exists.

In our differential meet-in-the-middle attacks, we dealt with the mixing layers by simply guessing our way past them. We expect significant improvements to the attacks are possible with more analysis of the mixing layers, particularly in terms of partial guessing of key material. We have spent far more time analyzing the core rounds than the unkeyed mixing layers, and so this is a good area for further research.

Attacking the symmetrically reduced version of the cipher with $k = 4$ apparently requires a better way of choosing inputs than the input structure we discuss above. We hope to find a better input structure, or a better property to push through all eight core rounds. Previous attempts to attack $k = 4$ have exceeded 2^{256} work, usually due to the huge plaintext requirements.

It may also be worthwhile to attack variants of MARS that cut off in the middle (after 12 or 16 rounds total); we have some ideas in this direction.

Finally, in future work, we hope to examine how the MARS key schedule functions with various reduced-round variants.

6 Acknowledgements

The "extended Twofish team" met for two week-long cryptanalysis retreats during Fall 1999, once in San Jose and again in San Diego. This paper is a result of those collaborations. Our analysis of MARS and Serpent has very much been a team effort, with everybody commenting on all aspects. The authors would like to thank Niels Ferguson, Mike Stay, David Wagner, and Doug Whiting for useful conversations and comments on these attacks, and for the great time we had together. The authors would also like to thank Susan Langford for helpful suggestions on one of the attacks, and Beth Friedman for proofreading the final paper.

References

[ABK98] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," NIST AES Proposal, Jun 98.

[Ada98] C. Adams, "The CAST-256 Encryption Algorithm," NIST AES Proposal, Jun 98.

[BBS99] E. Biham, A. Biryukov, and A. Shamir, "Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials," *Advances in Cryptology — EUROCRYPT '99 Proceedings*, Springer-Verlag, 1999, pp. 12–23.

[BCD+98] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, "MARS — A Candidate Cipher for AES," NIST AES Proposal, Jun 98.

[Bih94] E. Biham, "New Types of Cryptanalytic Attacks Using Related Keys," *Journal of Cryptology*, v. 7, n. 4, 1994, pp. 229–246.

[Bih95] E. Biham, "On Matsui's Linear Cryptanalysis," *Advances in Cryptology — EUROCRYPT '94 Proceedings*, Springer-Verlag, 1995, pp. 398–412.

[BS93] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[BW99] A. Biryukov and D. Wagner, "Slide Attacks," *Fast Software Encryption, 6th International Workshop*, Springer-Verlag, 1999.

[Dae95] J. Daemen, "Cipher and Hash Function Design," Ph.D. thesis, Katholieke Universiteit Leuven, Mar 95.

[DR98] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," NIST AES Proposal, Jun 98.

[HKM95] C. Harpes, G. Kramer, and J. Massey, "A Generalization of Linear Cryptanalysis and the Applicability of Matsui's Piling-up Lemma," *Advances in Cryptology — EUROCRYPT '95 Proceedings*, Springer-Verlag, 1995, pp. 24–38.

[HKR+98] C. Hall, J. Kelsey, V. Rijmen, B. Schneier, and D. Wagner, "Cryptanalysis of SPEED," *Selected Areas in Cryptography*, Springer-Verlag, 1998, to appear.

[HM97] C. Harpes and J. Massey, "Partitioning Cryptanalysis," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 13–27.

[KM96] L.R. Knudsen and W. Meier, "Improved Differential Attacks on RC5," *Advances in Cryptology — CRYPTO '96*, Springer-Verlag, 1996, pp. 216–228.

[Knu94a] L.R. Knudsen, "Block Ciphers — Analysis, Design, Applications," Ph.D. dissertation, Aarhus University, Nov 1994.

[Knu95b] L.R. Knudsen, "Truncated and Higher Order Differentials," *Fast Software Encryption, 2nd International Workshop Proceedings*, Springer-Verlag, 1995, pp. 196–211.

[KR96] J. Kilian and P. Rogaway, "How to Protect DES Against Exhaustive Key Search," *Advances in Cryptology — CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 252–267.

[KRR+98] L.R. Knudsen, V. Rijmen, R. Rivest, and M. Robshaw, "On the Design and Security of RC2," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 206–221.

[KRW99] L.R. Knudsen, M.B.J. Robshaw, and D. Wagner, "Truncated Differentials in Skipjack," *Advances in Cryptology — CRYPTO '99 Proceedings*, Springer-Verlag, 1999, 99. 165–180.

[KS00] J. Kelsey and B. Schneier, "Initial Cryptanalysis of the MARS Core," draft.

[KSW99] J. Kelsey, B. Schneier, and D. Wagner, "Key Schedule Weaknesses in SAFER+," *The Second Advanced Encryption Standard Candidate Conference*, 1999, pp. 155–167.

[LMM91] X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," *Advances in Cryptology — CRYPTO '91 Proceedings*, Springer-Verlag, 1991, pp. 17–38.

[LH94] S. Langford and M. Hellman, "Differential-Linear Cryptanalysis," *Advances in Cryptology — CRYPTO '94*, Springer-Verlag, 1994.

[Mat94] M. Matsui, "Linear Cryptanalysis Method for DES Cipher," *Advances in Cryptology — EUROCRYPT '93 Proceedings*, Springer-Verlag, 1994, pp. 386–397.

[NIST97a] National Institute of Standards and Technology, "Announcing Development of a Federal Information Standard for Advanced Encryption Standard," *Federal Register*, v. 62, n. 1, 2 Jan 1997, pp. 93–94.

[NIST97b] National Institute of Standards and Technology, "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)," *Federal Register*, v. 62, n. 117, 12 Sep 1997, pp. 48051–48058.

[NSA98] NSA, "Skipjack and KEA Algorithm Specifications," Version 2.0, National Security Agency, 29 May 1998.

[RRS+98] R. Rivest, M. Robshaw, R. Sidney, and Y.L. Yin, "The RC6 Block Cipher," NIST AES Proposal, Jun 98.

[Saa99] M. Saarinen, “A Note Regarding the Hash Function Use of MARS and RC6,” available online from `http://www.jyu.fi/ mjos/`, 1999.

[SK96] B. Schneier and J. Kelsey, “Unbalanced Feistel Networks and Block Cipher Design,” *Fast Software Encryption, 3rd International Workshop Proceedings*, Springer-Verlag, 1996, pp. 121–144.

[SKW+98] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, “Twofish: A 128-Bit Block Cipher,” NIST AES Proposal, Jun 98.

[SKW+99] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, *The Twofish Encryption Algorithm: A 128-bit Block Cipher*, John Wiley & Sons, 1999.

[Vau96] S. Vaudenay, “An Experiment on DES Statistical Cryptanalysis,” *3rd ACM Conference on Computer and Communications Security*, ACM Press, 1996, pp. 139–147.

[Wag99] D. Wagner, “The Boomerang Attack,” *Fast Software Encryption, 6th International Workshop*, Springer-Verlag, 1999, pp. 156–170.

Impossible Differential on 8-Round MARS' Core

Eli Biham[*] Vladimir Furman[†]

March 15, 2000

Abstract

MARS is one of the AES finalists. The up-to-date analysis of MARS includes the discovery of weak keys, and Biham's estimation that a 12-round variant of MARS is breakable. This estimation was partly founded based on a 7-round impossible differential of the core of MARS. However, no such attack was presented to-date. In this paper we present two new longer impossible differentials of 8 rounds.

1 Introduction

MARS[5] is a block cipher designed by IBM as a candidate for the Advanced Encryption Standard selection process, and was accepted as one of the five finalists.

The up-to-date analysis of MARS includes weak keys, and Biham's estimation that MARS reduced to 12 rounds can be attacked[2]. This estimate was partially based on the existence of a 7-round impossible differential of MARS[1] (see [3, 4, 6] for more details on attacks using impossible differential). In this paper we introduce two 8-round impossible differentials of MARS' core.

2 An 8-Round Impossible Differential

We denote binary numbers with a subscript b, and a 32-bit binary numbers whose all bits except of bit i are all zero, and only bit i is one by $\delta_i = 0^{31-i}1^1 0^i_b$ (i.e., `1<<i` in C). We also denote a string of 0's (and 1's) of variable lengths (including zero length) by 0^*_b (and 1^*_b) and the complement of a bit-value x by $\bar{x}$ ($\bar{x} = 1 - x$).

[*] Computer Science Department, Technion - Israel Institute of Technology, Haifa 32000, Israel. biham@cs.technion.ac.il, http://www.cs.technion.ac.il/~biham/.

[†] Computer Science Department, Technion - Israel Institute of Technology, Haifa 32000, Israel. vfurman@cs.technion.ac.il.

The 7-round impossible wordwise (truncated) differential of MARS is of the form

$$(0,0,0,X) \overset{3\ rounds}{\rightarrow} (Y,0,0,0) \overset{1\ round}{\not\rightarrow} (0,0,0,W) \overset{3\ rounds}{\rightarrow} (Z,0,0,0)$$

where W, X, Y, and Z are non-zero, all pairs with differences of the form $(0,0,0,X)$ must have differences of the form $(Y,0,0,0)$ after 3 rounds, and similarly the differences $(0,0,0,W)$ always cause differences $(Z,0,0,0)$ after 3 rounds. However, there are no pairs with differences $(Y,0,0,0)$ such that the differences become $(0,0,0,W)$ after one round.

We observe that an extension of this impossible differential shows that when $W=\delta_{31}$ the intermediate one-round impossible differential can be replaced by a two-round impossible differential $(Y,0,0,0) \overset{2\ rounds}{\not\rightarrow} (0,0,0,\delta_{31})$, for some values of Y, leading to the following 8-round impossible differential for some values of X

$$(0,0,0,X) \overset{3\ rounds}{\rightarrow} (Y,0,0,0) \overset{2\ rounds}{\not\rightarrow} (0,0,0,\delta_{31}) \overset{3\ rounds}{\rightarrow} (\delta_{31},0,0,0).$$

In the following we describe the 3-round differentials with probability 1. Then, we describe why the 2-round intermediate differential is impossible, and for which values of Y. The conjunction of the various differentials to the 8-round impossible differentials is described at the end of this section.

2.1 The 3-Round Differentials with Probability 1

We denote additive difference by Δ, and XOR-differences by Δ_{xor}. In every round of MARS' core, every single 32-bit input word B, C and D influences only one 32-bit output word (on A, B and C respectively). Thus if we take the input difference of one of the foregoing to be non-zero (e.g., $\Delta B \neq 0$) and all others including ΔA to be 0 (e.g., $\Delta A = \Delta C = \Delta D = 0$), then we receive the output difference with only one non-zero difference. In particular, if we take some input difference $(0,0,0,X)$ where X is non-zero, we get the difference $(0,0,X_1,0)$ for some non-zero X_1 after one round, then the difference becomes $(0,X_2,0,0)$ for some non-zero X_2 after the next round. Finally, the difference becomes $(Y,0,0,0)$ for some non-zero Y after the third round. In total we get a 3-round truncated differential $(0,0,0,X) \rightarrow (Y,0,0,0)$ with probability 1.

Note that, if the least significant bits of X have the form $1\underbrace{0..0}_{i}$ $(i \geq 0)$, then the least significant bits of Y have the same form. It follows from the fact that the least significant bits of such form are preserved in both additive and XOR differences.

In the particular case $X=\delta_{31}$ we always get $Y=\delta_{31}$: We start with the following difference $(0,0,0,\delta_{31})$, i.e., $\Delta A_0 = \Delta B_0 = \Delta C_0 = 0, \Delta D_0 = \delta_{31}$. Since $\Delta A_0 = 0$, the mixings to B, C, and D have zero differences. Since the difference

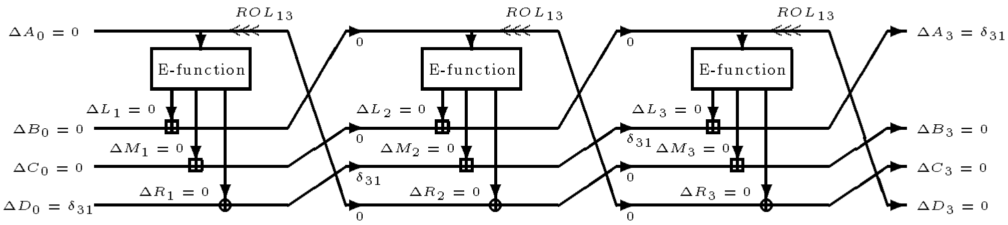


Figure 1: 3-round differential

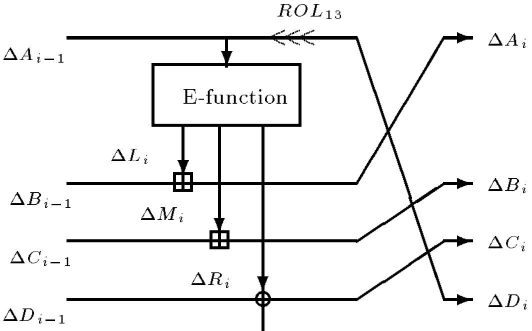


Figure 2: Round i in forward mode on MARS core

in ΔD is only in the most significant bit, this difference remains only in the most significant bit independently of whether the mixing operation is performed by addition or by XOR. Therefore, we get the difference $(\Delta A_1, \Delta B_1, \Delta C_1, \Delta D_1) = (0, 0, \delta_{31}, 0)$ after one round with probability 1. This can be repeated three times, and we get the difference $(\delta_{31}, 0, 0, 0)$ with probability one after 3 rounds, as shown in Figure 1. Notice, that this differential holds in all the rounds of the core including the forward mode, the backward mode and even on the boundary of both.

2.2 The 2-Round Impossible Differential

In this section we describe the 2-round impossible differential of MARS core.

Let $(\Delta A_0, \Delta B_0, \Delta C_0, \Delta D_0) = (Y, 0, 0, 0)$, where Y is an unknown value and $(\Delta A_2, \Delta B_2, \Delta C_2, \Delta D_2) = (0, 0, 0, \delta_{31})$. We want to find the values of Y that give impossible differential on a 2-round MARS core. We look for these values separately in the cases of forward and backward modes.

2.2.1 Forward Mode

Figure 2 outlines one round of the forward mode.

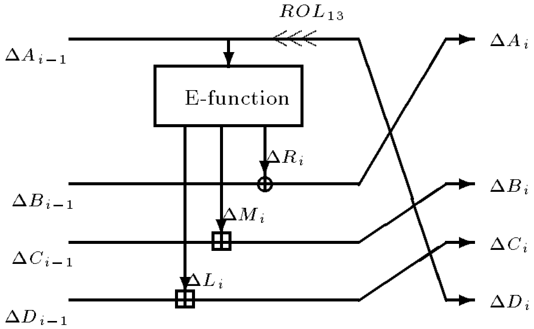


Figure 3: Round i backward mode on MARS core

- We know that $R_i = ((A_{i-1} <<< 13) \cdot K) <<< 10 = (D_i \cdot K) <<< 10$, where K is an unknown subkey. Because, the key used in this stage is odd and $\Delta D_2 = \delta_{31}$, we have that $\Delta_{xor} R_2 = \delta_9$.
- We have $\Delta_{xor} R_2 = \delta_9$ and $\Delta C_2 = \Delta_{xor} C_2 = 0$, so $\Delta_{xor} D_1 = \delta_9$. Thus, we receive $\Delta_{xor} A_0 = \delta_{28}$.
- $\Delta_{xor} A_0 = \delta_{28} \Rightarrow \Delta A_0 = aaa1\underbrace{0..0}_{28}{}_b$, where a is either 0 or 1 (i.e., $\Delta A_0 = \pm\delta_{28}$).

In total, we get that all values of Y, with possible exception of $\pm\delta_{28}$, give impossible differentials on a 2-round MARS core in the forward mode.

2.2.2 Backward Mode

The Figure 3 outlines the backward mode round.

- $\Delta D_2 = \delta_{31} \Rightarrow \Delta_{xor} D_2 = \delta_{31} \Rightarrow \Delta_{xor} A_1 = \delta_{18}$.
- $\Delta B_0 = 0 \Rightarrow \Delta_{xor} B_0 = 0$; Together with $\Delta_{xor} A_1 = \delta_{18}$ we get that $\Delta_{xor} R_1 = \delta_{18}$.
- $R_i = ((A_{i-1} <<< 13) \cdot K) <<< 10 = (D_i \cdot K) <<< 10$, so $\Delta_{xor}(D_i \cdot K) = \Delta_{xor} R_i >>> 10$. So $\Delta_{xor}(D_1 \cdot K) = \delta_{18} >>> 10 = \delta_8$, and $\Delta(D_1 \cdot K) = \Delta D_1 \cdot K = \pm\delta_8$. Because, the key used in this stage is odd, we have two important conclusions:
  1. ΔD_1 has $\underbrace{10..0}_{9}{}_b$ as a 9 least significant bits.
  2. We may look at this as $(\Delta D_1/2^8) \cdot (K \bmod 2^{24}) = \pm 1$. So the 24 least significant bits of the key are equal to the inverse of $\pm(\Delta D_1/2^8) \bmod 2^{24}$.

- On the other hand:

  $L_2 = (S[9 \text{ least significant bits of } (A_1 + K^+)] \oplus (R_2 >>> 5) \oplus R_2) <<< (5$ least significant bits of $R_2)$,

  where K^+ is an unknown subkey.

  - $\Delta_{xor} A_1 = \delta_{18}$ so the 9 least significant bits of ΔA_1 are 0, then $\Delta(9$ least significant bits of $(A_1 + K^+)) = 0$, so $\Delta S = 0$ and thus $\Delta_{xor} S = 0$.
  - As in forward mode, we get $\Delta_{xor} R_2 = \delta_9$, so $\Delta_{xor}(R_2 >>> 5) = \delta_4$.
  - $\Delta_{xor}(S \oplus (R_2 >>> 5) \oplus R_2) = \underbrace{0..0}_{22}1000010000_b$.
  - A variable rotation is performed on L_2 by a number of bits derived from the 5 least significant bits of R_2. Because $\Delta_{xor} R_2 = \delta_9$ both rotations are by the same number of bits (denoted by r_l), so we have:

    $$\Delta_{xor} L_2 = \underbrace{0..0}_{22}1000010000_b <<< r_l.$$

  - After the rotation, the result is always of the form:

    $$\Delta_{xor} L_2 = \underbrace{0..0}_{30-i-j}\ 1\,\underbrace{0..0}_{j}\,1\,\underbrace{0..0}_{i}{}_b,$$

    where $j = 4$ or 26, and $i = 0..30 - j$.
  - Thus we have $\Delta L_2 = \underbrace{b..b}_{30-i-j}\ \bar{a}\,\underbrace{a..a}_{j}\,1\,\underbrace{0..0}_{i}{}_b$, where a,b are unknown bit values.

- Because $\Delta L_2 + \Delta D_1 = \Delta C_2 = 0$, we have that $\Delta D_1 = \underbrace{\bar{b}..\bar{b}}_{30-i-j}\ a\,\underbrace{\bar{a}..\bar{a}}_{j}\,1\,\underbrace{0..0}_{i}{}_b$. But we know that ΔD_1 has $\underbrace{10..0}_{9}{}_b$ as the 9 least significant bits, so only a single possibility remains:

  $$\Delta D_1 = \underbrace{\bar{b}..\bar{b}}_{18}\,a\,\underbrace{\bar{a}..\bar{a}}_{4}\,1\,\underbrace{0..0}_{8}{}_b.$$

  Observation: ΔD_1 may have four possible values:

  - $\underbrace{0..0}_{18}1\underbrace{0..0}_{4}1\underbrace{0..0}_{8}{}_b$ and $\underbrace{1..1}_{18}0\underbrace{1..1}_{4}1\underbrace{0..0}_{8}{}_b$ (i.e., $\pm\underbrace{0..0}_{18}1\underbrace{0..0}_{4}1\underbrace{0..0}_{8}{}_b$).
  - $\underbrace{0..0}_{18}0\underbrace{1..1}_{4}1\underbrace{0..0}_{8}{}_b$ and $\underbrace{1..1}_{18}1\underbrace{0..0}_{4}1\underbrace{0..0}_{8}{}_b$ (i.e., $\pm\underbrace{0..0}_{19}\underbrace{1..1}_{4}1\underbrace{0..0}_{8}{}_b$).

We have two pairs of possible values for ΔD_1, and thus there are only two possible values (one for each pair) for the 24 least significant bits of the key used in first round for multiplication (according to the conclusion in the beginning of this section(2.2.2)). These key values in hexadecimal form are $f07c1f_x$ (for $\Delta D_1 = \pm\underbrace{0..0}_{18}1\underbrace{0..0}_{4}1\underbrace{0..0}_{8}{}_b$) and $ef7bdf_x$ (for $\Delta D_1 = \pm\underbrace{0..0}_{19}\underbrace{1..1}_{4}1\underbrace{0..0}_{8}{}_b$).

- It is known that sequences of the form 01^*1_b or of the form 10^*1_b in the additive difference (Δ) are translated to the sequence of the form either $100^*1^*1_b$ or 01^*1_b in the corresponding XOR difference (Δ_{xor})[1]. Thus we have two options:

  1. $\Delta_{xor}D_1 = \underbrace{0^*1^*}_{18}\underbrace{100^*1^*}_{5}1\underbrace{0..0}_{8}{}_b$
  2. $\Delta_{xor}D_1 = \underbrace{0^*1^*}_{18}\underbrace{01..1}_{5}1\underbrace{0..0}_{8}{}_b$

- $\Delta_{xor}A_0 = \Delta_{xor}D_1 >>> 13$, so there are two possible values for $\Delta_{xor}A_0$:

  1. $\Delta_{xor}A_0 = \underbrace{00^*1^*}_{4}1\underbrace{0..0}_{8}\underbrace{0^*1^*}_{18}1_b$
  2. $\Delta_{xor}A_0 = \underbrace{1..1}_{4}1\underbrace{0..0}_{8}\underbrace{0^*1^*}_{18}0_b$

- In the first case, $\Delta_{xor}A_0$ is odd, so the ΔA_0 is odd too, and we cannot show that this case is impossible. In the second case, $\Delta_{xor}A_0$ is even so the ΔA_0 is even too, and therefore we can divide this case in two sub-cases:

  1. There is at least one 1 in $\underbrace{0^*1^*}_{18}{}_b$, so we have 10_b as two least significant bits in $\Delta_{xor}A_0$ and ΔA_0. This sub-case is impossible (see Appendix A for a detailed proof).

[1] For checking this fact, look at different cases of such sequence with and without carry from previous bits. For example, we take $\Delta I = 10..01_b$, i.e., $I^1 - I^2 = 10..01_b$. Then either:

1. The least significant bit of I^1 is 1: then the least significant bit of I^2 must be 0, and thus there is no carry to the next bit. On the other hand, the next bit in the difference is 0. Combining these together we conclude that the next bit in I^1 and the next bit in I^2 must be equal. Continuing in this way we get that $\Delta_{xor}I = 10..01_b$.
2. The least significant bit of I^1 is 0: then the least significant bit of I^2 must be 1, and thus there is a carry to the next bit. On the other hand, the next bit in the difference is 0. Combining these together we conclude that the next bit in I^1 and the next bit in I^2 have different values. Continuing in this way we get that the corresponding bits in I^1 and in I^2 are different till either: 1) in some bit I^1 has 1 and in I^2 has 0, or 2) we reach the most significant bits with difference 1 and, due to existence of a carry from the previous bits, this bit in I^1 and I^2 must have the same value. So $\Delta_{xor}I$ is equal either to $100^*1^*11_b$ or to $01..11_b$.

2. There are no 1's in $\underbrace{0^*1^*}_{18}{}_b$, so $\Delta_{xor}A_0 = \underbrace{1..1}_{4}1\underbrace{0..0}_{27}{}_b$, and ΔA_0 has $1\underbrace{0..0}_{27}{}_b$ as 28 least significant bits. For this sub-case, we cannot show that it is impossible.

Thus, we have a 2-round impossible differential for any **even** Y whose 28 least significant bits are not $\underbrace{10..0}_{28}{}_b$. For other Y's we cannot say anything whether there exist impossible differentials. However, if the differentials are not impossible for some Y, then the 24 least significant bits of the multiplication key used in the first round of the differential are either $f07c1f_x$ or $ef7bdf_x$.

2.3 Conjunction to the 8-Round Impossible Differentials

We want now to check what values of X give the 8-round impossible differentials. We describe the two cases in which the two middle rounds work in forward mode and in backward mode.

For forward mode, we have a 2-round impossible differential for any value of Y, except of $\pm\delta_{28}$. Because in $(0,0,0,X) \stackrel{3\ rounds}{\rightarrow} (Y,0,0,0)$ the relation between X to Y passes through two additions and one exclusive-or operation, the 29 rightmost bits remains $1\underbrace{0..0}_{28}{}_b$ and the 3 most significant bits may get any value. So, we have the 8-round impossible differentials $(0,0,0,X) \overset{8\ rounds}{\not\rightarrow} (\delta_{31},0,0,0)$ for all X, except of those with $\underbrace{10..0}_{29}{}_b$ as the 29 least significant bits.

For backward mode, we have a 2-round impossible differential for any even Y, except of those with $\underbrace{10..0}_{28}{}_b$ as 28 least significant bits. As in forward mode, in $(0,0,0,X) \stackrel{3\ rounds}{\rightarrow} (Y,0,0,0)$ the 28 least significant bits remains $\underbrace{10..0}_{28}{}_b$ and the 4 most significant bits may get any value. So we have the 8-round impossible differentials $(0,0,0,X) \overset{8\ rounds}{\not\rightarrow} (\delta_{31},0,0,0)$ for any even X, except of those with $\underbrace{10..0}_{28}{}_b$ as the 28 least significant bits.

3 Another 8-Round Impossible Differential

There is another 8-round impossible differential on MARS' core:

$$(0,0,0,\delta_{31}) \stackrel{3\ rounds}{\rightarrow} (\delta_{31},0,0,0) \overset{3\ rounds}{\not\rightarrow} (0,0,X,\delta_{31}) \stackrel{2\ rounds}{\rightarrow} (Y,\delta_{31},0,0),$$

where the 3 middle round are in backward mode, and X,Y are non-zero values such that X must have $\underbrace{0..0}_{24}{}_b$ as the least significant bits, and the 8 most significant bits of X may have any value (except of all zeroes). Thus, as was shown in the previous section, Y must have $\underbrace{0..0}_{24}{}_b$ as the least significant bits, and the 8 most significant bits may have any value (except of all zeroes). The explanation for this differential is similar to the explanation described earlier.

Acknowledgments

We would like to acknowledge the work of Alon Becker and Eran Richardson who made the initial observations in this direction.

References

[1] A. Becker, E. Richardson, Course Project, 1998.

[2] E. Biham, *A note on Comparing the AES Candidates*, Second AES Conference, March 1999.

[3] E. Biham, A. Biryukov, A. Shamir, *Cryptanalysis of Skipjack Reduced o 31 Rounds Using Impossible Differentials*, LNCS, Advanced in Cryptology - Proceeding of EUROCRYPT'99, Springer-Verlag 1999.

[4] E. Biham, A. Biryukov, A. Shamir, *Miss in the Middle Attacks on IDEA, Khufu, and Khafre*, LNCS 1636, Fast Software Encryption, pp. 124-138, March 1999.

[5] C. Burwick, D. Coppersmith, E. C'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, *"MARS - A Candidate Cipher for AES "*, NIST AES Proposal, June 1998.

[6] L. Knudsen, *DEAL - A 128-bit Block Cipher*, NIST AES Proposal, June 1998.

A Impossible differential for Y, with 10_b as least significant bits, in backward mode on MARS core.

In this appendix we show that the sub-case of backward mode where $\Delta_{xor}A_0 = \underbrace{1..1}_{4}1\,\underbrace{0..0}_{8}\underbrace{0^*1^*}_{17}10_b$, mentioned in section 2.2.2, is impossible.

- As in forward mode $\Delta D_2 = \delta_{31} \Rightarrow \Delta_{xor}R_2 = \delta_9$.
- $\Delta_{xor}R_2 \oplus \Delta_{xor}B_1 = \Delta_{xor}A_2 = 0$, so $\Delta_{xor}B_1 = \Delta_{xor}R_2 = \delta_9$.
- $\Delta_{xor}B_1 = \delta_9 \Rightarrow \Delta B_1 = \underbrace{a..a}_{22}1\,\underbrace{0..0}_{9}{}_b$, where a is unknown bit value.
- $\Delta C_0 + \Delta M_1 = \Delta B_1 = \underbrace{a..a}_{22}1\,\underbrace{0..0}_{9}{}_b$. Because $\Delta C_0 = 0$, $\Delta M_1 = \underbrace{a..a}_{22}1\,\underbrace{0..0}_{9}{}_b$.
- $\Delta M_1 = \underbrace{a..a}_{22}1\,\underbrace{0..0}_{9}{}_b \Rightarrow \Delta_{xor}M_1 = \underbrace{0^*1^*}_{22}1\,\underbrace{0..0}_{9}{}_b$.
- We know that $M_i = (A_{i-1} + K) <<<$ (low 5 bits of $(R_i >>> 5)$). However, because $\Delta_{xor}R_1 = \delta_{18}$, both rotations are by the same number of bits (denoted r_m), and because $\Delta K = 0$ we have
$$\Delta M_1 = \Delta A_0 <<< r_m$$
or
$$\Delta A_0 = \Delta M_1 >>> r_m.$$
- We know that $\Delta_{xor}A_0 = \underbrace{1..1}_{4}1\,\underbrace{0..0}_{8}\underbrace{0^*1^*}_{17}10_b$. It gives us that $\Delta A_0 = \underbrace{x}_{4}\bar{a}\,\underbrace{a..a}_{9}\underbrace{z}_{17}10_b$, where x, z are unknown binary word and a is unknown bit value.
- The ΔA_0 has 10_b as 2 least significant bits, so the only one possibility for r_m to be 8. Thus $\Delta_{xor}(M_1 >>> 8) = \underbrace{0..0}_{8}\underbrace{0^*1^*}_{22}10_b$, and therefore, $\Delta(M_1 >>> 8) = \underbrace{b..b}_{8}\underbrace{y}_{22}10_b$, where b is an unknown bit value and y is unknown binary word.
- Now we have $\Delta(M_1 >>> 8) = \underbrace{b..b}_{8}\underbrace{y}_{22}10_b$ and $\Delta A_0 = \underbrace{x}_{4}\bar{a}\,\underbrace{a..a}_{9}\underbrace{z}_{17}10_b$. These must be equal. However, the bit 26th of the later differ than bit 27th, while bits 26th and 27th of the former are equal. This contradicts the fact that both values must be equal.

Preliminary Cryptanalysis of Reduced-Round Serpent

Tadayoshi Kohno[1]*, John Kelsey[2], and Bruce Schneier[2]

[1] Reliable Software Technologies
kohno@rstcorp.com
[2] Counterpane Internet Security, Inc.
{kelsey,schneier}@counterpane.com

Abstract. Serpent is a 32-round AES block cipher finalist. In this paper we present several attacks on reduced-round variants of Serpent that require less work than exhaustive search. We attack six-round 256-bit Serpent using the meet-in-the-middle technique, 512 known plaintexts, 2^{246} bytes of memory, and approximately 2^{247} trial encryptions. For all key sizes, we attack six-round Serpent using standard differential cryptanalysis, 2^{83} chosen plaintexts, 2^{40} bytes of memory, and 2^{90} trial encryptions. We present boomerang and amplified boomerang attacks on seven- and eight-round Serpent, and show how to break nine-round 256-bit Serpent using the amplified boomerang technique, 2^{110} chosen plaintexts, 2^{212} bytes of memory, and approximately 2^{252} trial encryptions.

1 Introduction

Serpent is an AES-candidate block cipher invented by Ross Anderson, Eli Biham, and Lars Knudsen [ABK98], and selected by NIST as an AES finalist. It is a 32-round SP-network with key lengths of 128 bits, 192 bits, and 256 bits. Serpent makes clever use of the bitslice technique to make it efficient in software. However, because of its conservative design and 32 rounds, Serpent is still three times slower than the fastest AES candidates [SKW+99].

In the Serpent submission document [ABK98], the authors give upper bounds for the best differential characteristics through the cipher. However, no specific attacks on reduced-round versions of the cipher are presented. In this paper we consider four kinds of attacks on reduced-round variants of Serpent: differential [BS93], boomerang [Wag99], amplified boomerang [KKS00], and meet-in-the-middle. To the best of our knowledge, these are the best published attacks against reduced-round versions of Serpent.[1]

The current results on Serpent are as follows (see Table 1):

1. A meet-in-the-middle attack on Serpent reduced to six rounds, requiring 512 known plaintext/ciphertext pairs, 2^{246} bytes of random-access memory, and work equivalent to approximately 2^{247} six-round Serpent encryptions.

[*] Part of this work was done while working for Counterpane Internet Security, Inc.

[1] Dunkelman cryptanalyzed a Serpent variant with a modified linear transformation in [Dun99].

Rounds	Key Size	Complexity			Comments
		[Data]	[Work]	[Space]	
—	—	—	—	—	no previous results
6	256	512 KP	2^{247}	2^{246}	meet-in-the-middle (§6)
6	all	2^{83} CP	2^{90}	2^{40}	differential (§3.2)
6	all	2^{71} CP	2^{103}	2^{75}	differential (§3.3)
6	192 & 256	2^{41} CP	2^{163}	2^{45}	differential (§3.4)
7	256	2^{122} CP	2^{248}	2^{126}	differential (§3.5)
8	192 & 256	2^{128} CPC	2^{163}	2^{133}	boomerang (§4.2)
8	192 & 256	2^{110} CP	2^{175}	2^{115}	amp. boomerang (§5.3)
9	256	2^{110} CP	2^{252}	2^{212}	amp. boomerang (§5.4)

KP — known plaintext, CP — chosen plaintext, CPC — chosen plaintext/ciphertext.

Table 1. Summary of attacks on Serpent. Work is measured in trial encryptions; space is measured in bytes.

2. A differential attack on Serpent reduced to six rounds, requiring 2^{83} chosen plaintexts, 2^{40} bytes of sequential-access memory, and work equivalent to approximately 2^{90} six-round Serpent encryptions.
3. A differential filtering attack on Serpent reduced to seven rounds, requiring 2^{122} chosen plaintexts, 2^{126} bytes of sequential-access memory, and work equivalent to approximately 2^{248} six-round Serpent encryptions.
4. A boomerang attack on Serpent reduced to eight rounds, requiring all 2^{128} plaintext/ciphertext pairs under a given key, 2^{133} bytes of random-access memory, and work equivalent to approximately 2^{163} eight-round Serpent encryptions.[2]
5. An amplified-boomerang key-recovery attack on Serpent reduced to eight rounds, requiring 2^{110} chosen plaintexts, 2^{115} bytes of random-access memory, and work equivalent to approximately 2^{175} eight-round Serpent encryptions.
6. An amplified-boomerang key-recovery attack on Serpent reduced to nine rounds, requiring 2^{110} chosen plaintexts, 2^{212} bytes of random-access memory, and work equivalent to approximately 2^{252} nine-round Serpent encryptions.

The remainder of this paper is organized as follows: First, we discuss the internals of Serpent and explain the notation we use in this paper. We then use differential, boomerang, and amplified boomerang techniques to break up to nine rounds of Serpent. Subsequently we discuss a six-round meet-in-the-middle attack on Serpent. We then discuss some observations on the Serpent key schedule. We conclude with a discussion of our results and some open questions.

[2] Because this eight-round boomerang attack requires the entire codebook under a single key, one can consider this attack a glorified distinguisher that also recovers the key.

2 Description of Serpent

In this document we consider only the bitsliced version of Serpent. The bitsliced and non-bitsliced versions of Serpent are functionally equivalent; the primary difference between the bitsliced and non-bitsliced versions of Serpent are the order in which the bits appear in the intermediate stages of the cipher. Full details of the bitsliced and non-bitsliced version of Serpent are in [ABK98].

2.1 The Encryption Process

Serpent is a 32-round block cipher operating on 128-bit blocks. In the bitsliced version of Serpent, one can consider each 128-bit block as the concatenation of four 32-bit words.

Let B_i represent Serpent's intermediate state prior to the ith round of encryption. Notice that $B_0 = P$ and $B_{32} = C$, where P and C are the plaintext and ciphertext, respectively.

Let K_i represent the 128-bit ith round subkey and let S_i represent the application of the ith round S-box. Let L be Serpent's linear transformation. Then the Serpent round function is defined as:

$$\begin{aligned}
X_i &\leftarrow B_i \oplus K_i \\
Y_i &\leftarrow S_i(X_i) \\
B_{i+1} &\leftarrow L(Y_i) \qquad i = 0, \ldots, 30 \\
B_{i+1} &\leftarrow Y_i \oplus K_{i+1} \; i = 31
\end{aligned}$$

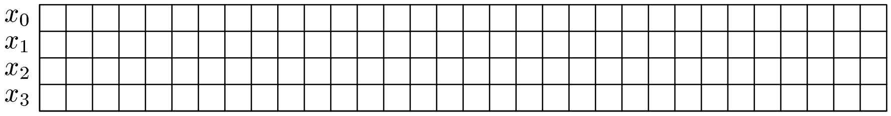
Serpent uses eight S-boxes $S_0, \ldots, S_7$. The indices to S are reduced modulo 8; i.e., $S_0 = S_8 = S_{16} = S_{24}$. The Serpent S-boxes take four input bits and produce four output bits. Consider the application of an S-box S_i to the 128 bit block X_i. Serpent first separates X_i into four 32-bit words x_0, x_1, x_2, and x_3. For each of the 32-bit positions, Serpent constructs a nibble from the corresponding bit in each of the four words, with the bit from x_3 being the most significant bit. Serpent then applies the S-box S_i to the constructed nibble and stores the result in the respective bits of $Y_i = (y_0, y_1, y_2, y_3)$.

The linear transform L on $Y_i = (y_0, y_1, y_2, y_3)$ is defined as

$$\begin{aligned}
y_0 &\leftarrow y_0 \lll 13 \\
y_2 &\leftarrow y_2 \lll 3 \\
y_1 &\leftarrow y_0 \oplus y_1 \oplus y_2 \\
y_3 &\leftarrow y_2 \oplus y_3 \oplus (y_0 \ll 3) \\
y_1 &\leftarrow y_1 \lll 1 \\
y_3 &\leftarrow y_3 \lll 7 \\
y_0 &\leftarrow y_0 \oplus y_1 \oplus y_3 \\
y_2 &\leftarrow y_2 \oplus y_3 \oplus (y_1 \ll 7) \\
y_0 &\leftarrow y_0 \lll 5 \\
y_2 &\leftarrow y_2 \lll 22 \\
B_{i+1} &\leftarrow (y_0, y_1, y_2, y_3)
\end{aligned}$$

where $\lll$ denotes a left rotation and $\ll$ denotes a left shift.

When discussing the internal state of the Serpent, we will often refer to diagrams such as



where X_i is the internal state under inspection and $X_i = (x_0, x_1, x_2, x_3)$. As suggested by this diagram, we will occasionally refer to an active S-box as a "column."

2.2 The Key Schedule

Serpent's key schedule can accept key sizes up to 256 bits. If a 256-bit key is used, Serpent sets the eight 32-bit words $w_{-8}, w_{-7}, \dots, w_{-1}$ to the key. If not, the key is converted to a 256-bit key by appending a '1' bit followed by a string of '0's.

Serpent computes the prekeys $w_0, w_1, \dots, w_{131}$ using the recurrence

$$w_i \leftarrow (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) \lll 11$$

where ϕ is `0x9e3779b9`.

Serpent then computes the 128-bit subkeys K_j by applying an S-box to the prekeys $w_{4j}, \dots, w_{4j+3}$:

$$\begin{aligned}
K_0 &\leftarrow S_3(w_0, w_1, w_2, w_3) \\
K_1 &\leftarrow S_2(w_4, w_5, w_6, w_7) \\
K_2 &\leftarrow S_1(w_8, w_9, w_{10}, w_{11}) \\
K_3 &\leftarrow S_0(w_{12}, w_{13}, w_{14}, w_{15}) \\
K_4 &\leftarrow S_7(w_{16}, w_{17}, w_{18}, w_{19}) \\
&\vdots \\
K_{31} &\leftarrow S_4(w_{124}, w_{125}, w_{126}, w_{127}) \\
K_{32} &\leftarrow S_3(w_{128}, w_{129}, w_{130}, w_{131})
\end{aligned}$$

3 Differential Cryptanalysis

Differential cryptanalysis, first publicly discussed by Biham and Shamir [BS93], is one of the most well-known and powerful cryptanalytic techniques. Although the original Serpent proposal provided theoretical upper bounds for the highest probability characteristics through reduced-round Serpent variants [ABK98], the Serpent proposal did not present any empirical results describing how successful differential cryptanalysis would be against Serpent in practice.

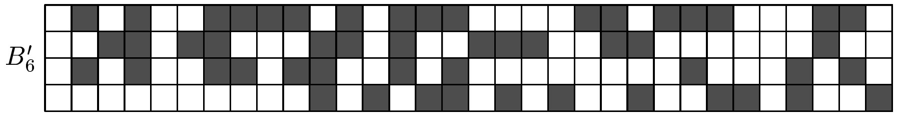
In this section we consider actual differential attacks against reduced-round Serpent variants. Although there may exist other high-probability differentials through several rounds of Serpent, we focus on a particular five-round characteristic, $B'_1 \to Y'_5$, with probability $p = 2^{-80}$. This characteristic spans Serpent's second through sixth rounds (rounds $i = 1, \ldots, 5$). For completeness, this characteristic is illustrated in Appendix A.1. Notationally, we use X' to represent the XOR difference between two values X and X^*.

3.1 Basic Six-Round Differential Attack

We can use the above-mentioned five-round, probability 2^{-80}, characteristic to attack rounds one through six of 192- and 256-bit Serpent.

To sketch our attack: we request 2^{82} plaintext pairs with an input difference B'_1. For each last round subkey guess, we initialize a count variable to zero. Then, for each unfiltered pair, we peel off the last round and look for our expected output difference from the fifth round. If we observe our expected output difference, we increment our counter. If we count three or more right pairs, we note this subkey as likely to be correct.

If we apply the linear transformation L to the intermediate difference Y'_5 (Appendix A.1), we get the following expected input difference to the sixth round:



We can immediately identify all but approximately 2^{-47} of our ciphertext pairs as wrong pairs because their differences B'_7 cannot correspond to our desired difference B'_6.

After filtering we are left with approximately 2^{35} ciphertext pairs. Our attack thus requires approximately $2^{36} \times 2^{116}$ partial decryptions, or work equivalent to approximately 2^{150} six-round Serpent encryptions. If we retain only our unfiltered ciphertext pairs, this attack requires approximately 2^{40} bytes of sequential memory. The signal-to-noise ratio of this attack is 2^{83}.

3.2 Improved Six-Round Differential Attack

By counting on fewer than 116 bits of the last round subkey, we can considerably improve the six-round differential attack in the previous section. For example, if we count on two sets of 56 bits, our work is reduced to about 2^{90} Serpent six-round encryptions. This allows us to break six rounds of 128-, 192-, and 256-bit Serpent using less work than exhaustive search.

3.3 Bypassing the First Round

We can use structures to bypass the first round of our five-round characteristic $B'_1 \rightarrow Y'_5$. This gives us an attack that requires fewer chosen plaintexts but more work than the attack in Section 3.2. In this attack we use the four-round, probability 2^{-67}, characteristic $B'_2 \rightarrow Y'_5$. We request 2^{47} blocks of 2^{24} plaintexts such that each block varies over all possible inputs to the active S-boxes in B'_1. This gives us 2^{70} pairs with our desired input difference to the second round. We expect eight pairs with our desired difference Y'_5.

We can mount the attack in Section 3.2 by looking for the last round subkey suggested seven or more times. In this attack we must consider a total of 2^{94} possible ciphertext pairs. As with Section 3.2, we can immediately identify all but 2^{-47} of these pairs as wrong pairs. This attack requires work equivalent to approximately 2^{102} Serpent six-round encryptions and approximately 2^{75} bytes of random-access memory.

3.4 Additional Six-Round Differential Attack

We can modify our basic six-round differential attack by guessing part of the last round subkey and looking at the eight passive S-boxes in B'_5. In order to do this, we must guess 124 bits of the last round subkey.

In this attack we request 2^{40} chosen-plaintext pairs with our input difference B'_1. This gives us 2^9 pairs with difference B'_5 entering the fifth round. For a correct 124-bit last round subkey guess, we expect to count 2^9 pairs with passive S-boxes in Y'_5 corresponding to the passive S-boxes in B'_5. For an incorrect last round subkey guess, the number of occurrences of pairs with passive differences in our eight target S-boxes is approximately normal with mean 2^8 and standard deviation 2^4. Since 2^9 is 16 standard deviations to the right of 2^8, we expect no false positives.

This attack requires 2^{45} bytes of sequential memory and work equivalent to approximately 2^{163} Serpent six-round encryptions.

3.5 Seven-Round Differential Filtering Attack

We can use our filtering scheme in Section 3.1 to distinguish six rounds of Serpent from a random permutation. In this distinguishing attack we request 2^{121} plaintext pairs with our desired input difference B'_1. We expect approximately 2^{41} right pairs. Since our filter passes ciphertext pairs with a probability 2^{-47}, we expect approximately $2^{74} + 2^{41}$ ciphertext pairs to pass our filter.

In a random permutation, the number of unfiltered pairs is approximately a normal distribution with mean 2^{74} and standard deviation 2^{37}. Since $2^{74} + 2^{41}$ is 16 standard deviations to the right of the random distribution's mean of 2^{74}, we can distinguish six-round Serpent from a random permutation. For a random distribution, the probability of observing $2^{74} + 2^{41}$ or more unfiltered pairs is approximately 2^{-190}.

We can extend this six-round distinguishing attack to a seven-round key recovery attack on rounds one through seven by guessing the entire last round subkey K_8 and performing our six-round distinguishing attack. This attack requires approximately 2^{126} bytes of sequential memory 2^{248} Serpent seven-round encryptions.

4 Boomerang Attacks

4.1 Seven-Round Boomerang Distinguisher

In addition to being able to perform traditional differential attacks against Serpent, we can also use Wagner's boomerang attack [Wag99] to distinguish seven rounds of Serpent from a random permutation.

Let us consider a seven-round variant of Serpent corresponding to the second through eighth rounds of the full 32-round Serpent (i.e., rounds $i = 1, \ldots, 7$). Call the first four rounds of this seven-round Serpent E_0 and call the final three rounds E_1. Our seven-round Serpent is thus $E = E_1 \circ E_0$. We can now apply the boomerang technique to this reduced-round Serpent.

Notice that if we only consider the first four rounds of the five-round characteristic in Appendix A.1, we have a four-round characteristic $B'_1 \to Y'_4$ through E_0 with probability 2^{-31}. Also notice that there exist three-round characteristics through E_1 with relatively high probability. Appendix A.2 illustrates one such characteristic, $B'_5 \to Y'_7$, with probability 2^{-16}.

To use the terminology in [Wag99], let $\Delta = B'_1$, let $\Delta^* = Y'_4$, let $\nabla = Y'_7$ and let $\nabla^* = B'_5$. We then use $\Delta \to \Delta^*$ as our differential characteristic for E_0 and $\nabla \to \nabla^*$ as our differential characteristic for E_1^{-1}.

In the boomerang distinguishing attack, we require approximately $4 \cdot 2^{94}$ adaptive-chosen plaintext/ciphertext queries, or approximately 2^{94} quartets P, P', Q, and Q' and their respective ciphertexts C, C', D, and D'. More specifically, in our distinguishing attack we request the ciphertext C and C' for about 2^{94} plaintexts P and P' where $P \oplus P' = \Delta$. From C and C' we compute the ciphertexts $D = C \oplus \nabla$ and $D' = C' \oplus \nabla$. We then apply the inverse cipher to D and D' to obtain Q and Q'. For any quartet P, P', Q, and Q', we expect the combined properties $P \oplus P' = Q \oplus Q' = \Delta$ and $C \oplus D = C' \oplus D' = \nabla$ to hold with probability 2^{-94}.

4.2 Eight-Round Boomerang Key Recovery Attack

We can extend our seven-round boomerang distinguisher to an eight-round key recovery attack on 192- and 256-bit Serpent reduced to rounds $i = 1, \ldots, 8$ (or rounds $i = 9, \ldots, 16$ or rounds $i = 17, \ldots, 24$). The basic idea is that we peel off the last round by guessing the last round subkey and look for our property in the preceding seven rounds.

A difficulty arises because the boomerang attack makes adaptive chosen plaintext *and* ciphertext queries. Suppose we encrypt P and P' to get C and

C'. To get D and D', we must peel off one round from each ciphertext C and C', XOR the result with ∇, and then re-encrypt the last round with the guessed subkey. To do this, we will have to guess the 68 bits of the last round subkey corresponding to the 17 active S-boxes of B'_8. Assume we consider 2^{94} plaintext pairs P and P'. For each of these pairs, we will have to compute 2^{68} different pairs Q and Q' (for each of the 2^{68} possible last round subkeys). Unfortunately, this means we will likely end up working with the entire codebook of all 2^{128} possible plaintext/ciphertext pairs.

If we are willing to work with the entire codebook of 2^{128} plaintexts and ciphertexts, then we can extract the last round subkey in the following manner. We request the ciphertexts C and C' of 2^{96} plaintext pairs with an input difference Δ. Then for each of our 2^{68} possible last round subkeys and for each of our 2^{96} ciphertext pairs, we compute the boomerang ciphertexts D and D'. We then request the plaintexts Q and Q' corresponding to these ciphertexts. If we correctly guess the last round subkey, we should expect to see the plaintext difference $Q \oplus Q' = \Delta$ with probability 2^{-94}. That is, for the correct subkey we should expect to see the difference $Q \oplus Q' = \Delta$ approximately four times. (Or, put yet another way, if we guess the correct subkey, we should generate about four right quartets.)

This attack requires $2^{68} \times 2^{97}$ partial decryptions and encryptions, or approximately 2^{163} eight-round Serpent encryptions. This attack also requires access to the entire codebook, and thus 2^{128} plaintexts and 2^{133} bytes of random-access memory.

5 Amplified Boomerang Attacks

In [KKS00] we introduced a new class of cryptanalytic attacks which we call "amplified boomerangs." Amplified boomerang attacks are similar to traditional boomerang attacks but require only chosen plaintexts. The chosen-plaintext–only requirement makes the amplified boomerang attacks more practical than the traditional boomerang attacks in many situations. In [KKS00] we describe a seven-round boomerang amplifier distinguishing attack and an eight-round boomerang amplifier key recovery attack requiring 2^{113} chosen plaintext pairs, 2^{119} bytes of random-access memory, and roughly 2^{179} Serpent eight-round encryptions.

5.1 Amplified Seven-Round Distinguisher

In this section we review the seven-round amplified boomerang distinguishing attack presented in [KKS00]. We request 2^{112} plaintext pairs with our input difference Δ. After encrypting with the first half of the cipher E_0, we expect roughly 2^{81} pairs to satisfy the first characteristic $\Delta \to \Delta^*$. There are approximately 2^{161} ways to form quartets using these 2^{81} pairs. We expect there to be approximately 2^{33} quartets (Y_4^0, Y_4^1) and (Y_4^2, Y_4^3) such that $Y_4^0 \oplus Y_4^2 = \nabla^*$. However, because (Y_4^0, Y_4^1) and (Y_4^2, Y_4^3) are right pairs for the first half of the cipher, and $Y_4^0 \oplus Y_4^1 = Y_4^2 \oplus Y_4^3 = \Delta^*$, we have that $Y_4^1 \oplus Y_4^3$ must also equal ∇^*. In effect, the randomly occurring difference between Y_4^0 and Y_4^2 has been "amplified" to include Y_4^1 and Y_4^3.

At the input to E_1 we expect approximately 2^{33} quartets with a difference of (∇^*, ∇^*) between the pairs. This gives us approximately two quartets after the seventh round with an output difference of (∇, ∇) across the pairs. We can identify these quartets by intelligently hashing our original ciphertext pairs with our ciphertext pairs XORed with (∇, ∇) and noting those pairs that collide. For a random distribution, the probability of observing a single instance of our cross-pair difference (∇, ∇) is approximately 2^{-33}.

5.2 Amplified Eight-Round Key Recovery Attack

In [KKS00] we extended the previous distinguishing attack to an eight-round key-recovery attack on rounds one through eight of Serpent requiring 2^{113} chosen-plaintext pairs, 2^{119} bytes of random-access memory, and work equivalent to approximately 2^{179} eight round Serpent encryptions. In this attack we guess 68 bits of Serpent's last round key K_9. For each key guess, we peel off the last round and perform the previous distinguishing attack.

5.3 Experimental Improvements to the Eight-Round Attack

We can improve our eight-round boomerang amplifier attack by observing that we do not need to restrict ourselves to using only one specific cross-pair difference (∇^*, ∇^*) after E_0. That is, rather than considering only pairs of pairs with a cross-pair difference of (∇^*, ∇^*) after E_0, we can use pairs of pairs with a cross-pair difference of (x, x) after E_0, for *any* x, provided that both pairs follow the characteristic $x \rightarrow \nabla$ through E_1 with sufficiently high probability.

Experimentally, we find that $\sum_x Pr[x \rightarrow \nabla \text{ through } E_1]^2$ is approximately 2^{-23}.[3] Consequently, if we request 2^{109} chosen-plaintext pairs with our input difference Δ to E_0, we should expect approximately 16 pairs of pairs with a cross-pair difference of (∇, ∇) after E_1. This reduces the work of our attack in Section 5.2 to approximately 2^{175} eight-round Serpent encryptions. As noted in [Wag99], this observation can also be used to improve the standard boomerang attack.

5.4 Amplified Nine-Round Key Recovery Attack

We can further extend the above eight-round attack to break nine rounds of 256-bit Serpent using less work than exhaustive search. To do this, let us consider a nine-round Serpent variant corresponding to rounds zero through eight of

[3] We generated 2^{28} pairs of ciphertext pairs with a cross-pair difference (∇, ∇). We decrypted each pair through E_1^{-1} and counted the number of pairs with a cross pair difference (x, x) for any x. We observed 35 such pairs of pairs.

Serpent. Let us still refer to rounds one through four as E_0 and rounds five through seven as E_1.

If we apply the inverse linear transformation to Δ we get

Y_0'	□	□	□	□	□	■	■	□	□	□	□	■	□	■	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□	□	□	□
	■	□	□	■	□	■	□	□	□	□	□	□	□	□	□	■	□	■	□	□	■	■	■	□	□	■	□	■	■	□	■	■
	□	□	■	■	□	□	■	□	□	□	□	□	□	□	□	□	■	□	■	□	■	□	□	■	■	■	□	■	■	□	■	□
	□	□	□	■	□	□	■	■	□	□	■	□	□	■	□	■	□	■	□	□	□	□	□	□	■	■	□	□	□	□	□	■

where Y_0' has 24 active S-boxes. We request 2^{14} blocks of 2^{96} chosen plaintexts such that each block varies over all the possible inputs to the active S-boxes in Y_0'. This gives us 2^{109} pairs with our desired difference Δ into E_0 and 16 pairs of pairs with a cross-pair difference (∇, ∇) after E_1. In order to identify our amplified boomerang, we must guess 96 bits of the first round subkey K_0 and 68 bits of the last round subkey K_9.

We first guess the 96 bits of K_0 corresponding to the 24 active S-boxes in Y_0'. For each 96 bit key guess and for each plaintext P, we encrypt P one round to Y_0. We store P with satellite data Y_0 in HASH0$[K_0]$ and we store Y_0 with satellite data P in HASH1$[K_0]$. This step takes approximately 2^{212} bytes of random-access memory and work equivalent to 2^{203} Serpent eight-round encryptions.

Next, for each 68-bit key guess of K_9, we want to establish a list of all pairs (P^0, P^2) that have difference ∇ as the output of the eighth round. To do this, for each ciphertext C^0, we decrypt up one round to X_8^0, compute $X_8^2 = X_8^0 \oplus B_8'$, and store (X_8^0, X_8^2) or (X_8^2, X_8^0) in a hash table (where the order of X_8^0 and X_8^2 depends on whether X_8^0 is less than X_8^2). The satellite data in our hash table entry includes the plaintext P^0 corresponding to C^0. If a collision occurs in our hash table, we have found two plaintexts P^0 and P^2 that have our desired difference ∇ after the eighth round. We store these pairs (P^0, P^2) in LIST2$[K_9]$ and HASH2$[K_9]$. This step takes approximately 2^{184} bytes of random-access memory and work equivalent to 2^{175} Serpent eight-round encryptions.

The following algorithm counts the number of occurrences of our boomerang amplifier through $E_1 \circ E_0$. This algorithm can be thought of as sending a boomerang from the ciphertext to the plaintext and back again:

```
for each 96-bit subkey guess of $K_0$ do
  for each 68-bit subkey guess of $K_9$ do
    $count \leftarrow 0$
    for each pair $(P^0, P^2)$ in LIST2$[K_9]$ do
      lookup $Y_0^0$, $Y_0^2$ corresponding to $P^0$, $P^2$ in HASH0$[K_0]$
      $Y_0^1 \leftarrow Y_0^0 \oplus Y_0'$, $Y_0^3 \leftarrow Y_0^2 \oplus Y_0'$
      lookup $P^1$, $P^3$ corresponding to $Y_0^1$, $Y_0^3$ in HASH1$[K_0]$
      if $(P^1, P^3)$ in HASH2$[K_9]$ then
        $count \leftarrow count + 1$
    if $count \geq 15$ then
      save key guess for $K_0$, $K_9$
```

For each subkey guess guess of K_9, we expect LIST2$[K_9]$ will contain approximately $2^{219} \times 2^{-128} = 2^{91}$ pairs. Consequently, we expect the inner loop of the above algorithm to execute 2^{255} times. This attack requires work equivalent to approximately 2^{252} Serpent nine-round encryptions.

6 Meet-in-the-Middle Attacks

Although not as powerful as our previous attacks, we can use the meet-in-the-middle technique to attack six-round Serpent. In the meet-in-the-middle attack, we try to determine the value of a set of intermediate bits in a cipher by guessing key bits from both the plaintext and ciphertext sides. The attack looks for key guesses that match on the predicted values of the intermediate bits.

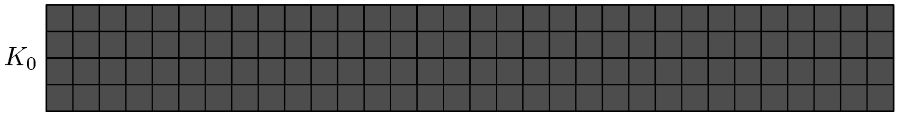
We did a computer search for the best meet-in-the-middle attacks that isolate a set of bits in one column of an intermediate state of Serpent. Table 2 summarizes our results. Although we can also use the meet-in-the-middle technique to predict bits in more than one column of an intermediate state of Serpent, doing so requires additional key guesses and is thus undesirable.

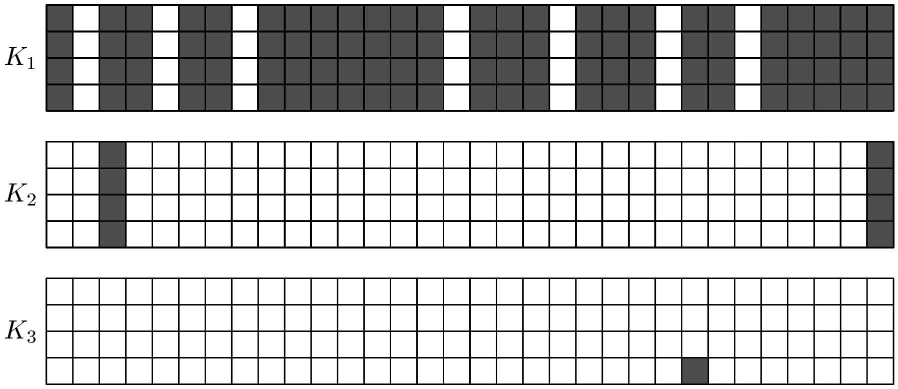
Rounds	b	s	Key guess from top	Key guess from bottom
6	1	B_3	236	239
5	2	B_2	152	223
5	3	B_2	176	224
5	4	B_2	204	225
6	1	X_3	237	238
5	2	X_2	154	221
5	3	X_2	179	221
5	4	X_2	208	221
5	1	Y_2	200	104
5	2	Y_2	200	178
5	3	Y_2	208	198
5	4	Y_2	208	221

Table 2. Meet-in-the-middle requirements to determine b intermediate bits of internal state s in a given number round Serpent variant.

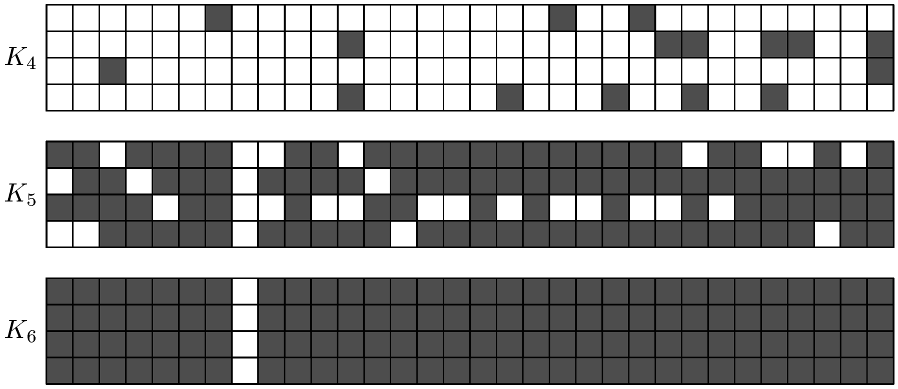
The clearest way to illustrate the meet-in-the-middle attack on Serpent is through diagrams similar to those used in Section 3 and Appendix A. The plaintext in this attack on six-round Serpent is B_0 and the ciphertext is B_6. The bit we are trying to predict is the eighth most significant bits of x_3 where x_3 is the fourth word of X_3, $X_3 = (x_0, x_1, x_2, x_3)$.

The 237 key bits guessed from the plaintext side are





and the 238 key bits guessed from the ciphertext side are



where the shaded cells denote the bits we guess.

The attack proceeds as follows. We obtain 512 known plaintexts and their corresponding ciphertexts. For each plaintext key guess, we compute the target bit of X_3 for each of our 512 plaintexts. We concatenate these bits for each plaintext into a 512-bit value. We then store this 512-bit value, along with the associated key guess, in a hash table.

For each ciphertext key guess, we proceed along the same lines and compute the target bit of X_3 for each of our 512 ciphertexts. We concatenate these bits for each ciphertext into a 512-bit value and look for this value in our hash table. If we find such a value, then the plaintext and ciphertext keys suggested by the match will likely be correct. This attack requires approximately 2^{246} bytes of random-access memory and work equivalent to 2^{247} six-round encryptions.

7 Key Schedule Observations

This section addresses some observations we have about the Serpent key schedule. We currently do not know of any cryptanalytic attacks that use these observations.

As described in Section 2, the prekeys $w_0, w_1, \ldots, w_{131}$ are computed using the recurrence

$$w_i \leftarrow (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) \lll 11 \tag{1}$$

where $w_{-8}, \ldots, w_{-1}$ is the initial 256 bit master key. If we ignore the rotation and the internal XOR with ϕ and i, we get the linear feedback construction

$$w_i \leftarrow w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \tag{2}$$

Let us now consider two keys K and K^* that have a difference $K' = K \oplus K^*$. The prekeys for K and K^* expand to $w_0, \ldots, w_{131}$ and $w_0^*, \ldots, w_{131}^*$, respectively. By virtue of Equation 2, the prekey differences for K' can be computed using the recurrence

$$w_i' = w_i \oplus w_i^* = w_{i-8}' \oplus w_{i-5}' \oplus w_{i-3}' \oplus w_{i-1}' \tag{3}$$

for $i = 0, \ldots, 131$. If we use the original recurrence (Equation 1) to compute the prekeys rather than Equation 2, the recurrence for w_i' becomes

$$w_i' = (w_{i-8}' \oplus w_{i-5}' \oplus w_{i-3}' \oplus w_{i-1}') \lll 11 \tag{4}$$

for $i = 0, \ldots, 131$.

For any key K, the ith round subkey K_i is computed from the four prekeys $w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}$. The same can be said for the key K^*. If for any given round i the four prekeys for K are equivalent to the corresponding four prekeys for K^*, then the subkeys K_i and K_i^* will be equivalent; this occurs when the prekey differences $w_{4i}', w_{4i+1}', w_{4i+2}', w_{4i+3}'$ are zero.

Let us now observe some situations where the prekey differences for the ith round subkey are zero. As a simple example, let us consider Figure 1. The shaded cells in Figure 1 depict prekeys that are different for K and K^*. The unshaded areas are equivalent between the keys. Notice that six out of the 33 128-bit subkeys are equivalent.

There is a heavy restriction on Figure 1: all the differences must be the same. That is, when Equation 2 is used for the prekey computation, it must be that $w_{-5}' = w_{-3}' = w_{-1}' = \cdots = w_{127}' = k$ for some constant k. If we consider the original prekey recursion (Equation 4), this example works only when $k =$ `0xFFFFFFFF`. Furthermore, when the non-zero prekey differences are `0xFFFFFFFF`, six out of 33 subkeys are equivalent and five out of 33 subkeys have complementary prekeys.

8 Conclusions

In this paper we consider several attacks on Serpent. We show how to use differential, boomerang, and amplified boomerang techniques to recover the key for Serpent up to nine rounds. We also show how to break six rounds of Serpent

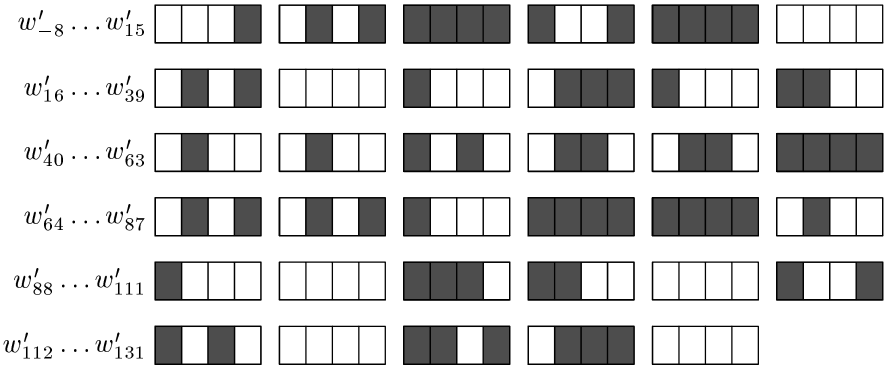


Fig. 1. Difference propagation in the key schedule when $w'_{-5} = w'_{-3} = w'_{-1} =$ `0xFFFFFFFF`.

using a meet-in-the-middle attack. We then provide key schedule observations that may someday be used as the foundation for additional attacks.

Although these attacks do not come close to breaking the full 32-round cipher, we feel that these results are worth reporting for several reasons. Specifically, the results and observations in this paper provide a starting point for additional research on Serpent. These results also provide a security reference point for discussions about modifying the number of rounds in Serpent.

In conjunction with the previous observation, we would like to point out that there are several avenues for further research. Although our current paper addresses differential attacks against Serpent, we have not yet tried linear and differential-linear attacks. We are also attempting to mount additional boomerang variants against Serpent. We expect that all these attacks, while quite capable of breaking reduced-round versions of Serpent, will fail to break the entire 32-round Serpent. In order to break a substantial portion of Serpent's 32 rounds, we suspect that entirely new attacks may need to be invented.

9 Acknowledgements

The "extended Twofish team" met for two week-long cryptanalysis retreats during Fall 1999, once in San Jose and later in San Diego. This paper is a result of those collaborations. Our analysis of Serpent has very much been a team effort, with everybody commenting on all aspects. The authors would like to thank Niels Ferguson, Chris Hall, Mike Stay, David Wagner, and Doug Whiting for useful conversations and comments on these attacks, and for the great time we had together.

References

[ABK98] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," NIST AES Proposal, 1998.

[BS93] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[Dun99] O. Dunkelman, "An Analysis of Serpent-p and Serpent-p-ns," rump session, *Second AES Candidate Conference*, 1999.

[KKS00] J. Kelsey, T. Kohno, and B. Schneier, "Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent," *Fast Software Encryption, 7th International Workshop*, to appear.

[SKW$^+$99] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Performance Comparison of the AES Submissions," *Second AES Candidate Conference*, 1999.

[Wag99] D. Wagner, "The Boomerang Attack," *Fast Software Encryption, 6th International Workshop*, Springer-Verlag, 1999.

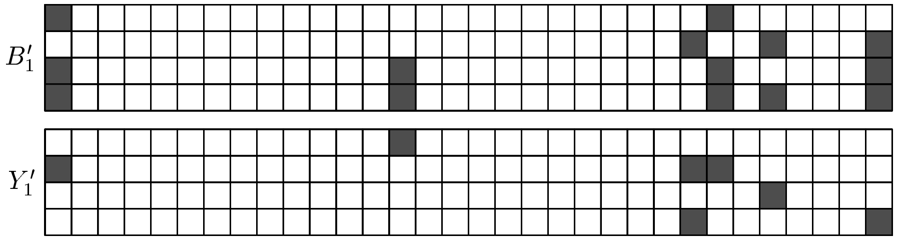
A Differential Characteristics

A.1 Five-Round Characteristic

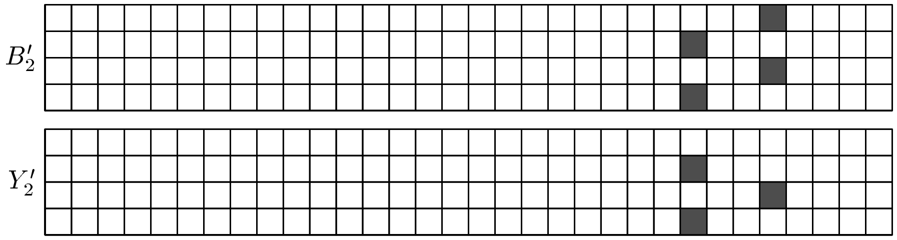
The following is an example of a five-round differential characteristic with probability $p = 2^{-80}$. We used this characteristic in Section 3. This characteristic passes between rounds $i = 1 \bmod 8$ and $i = 5 \bmod 8$. We used only the first four rounds of this five-round characteristic for our boomerang attack in Section 4.

We illustrate this characteristic by showing five one-round characteristics that can be connected with the Serpent linear transformation L. The shaded bits in the figures denote differences in the pairs. We feel that these figures provide an intuitive way to express Serpent's internal states.

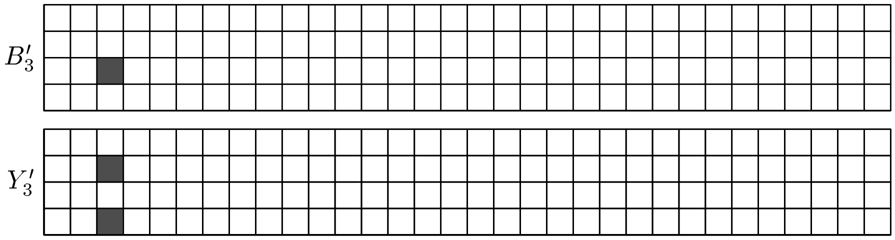
The first-round characteristic, $B_1' \to Y_1'$, has probability 2^{-13}:



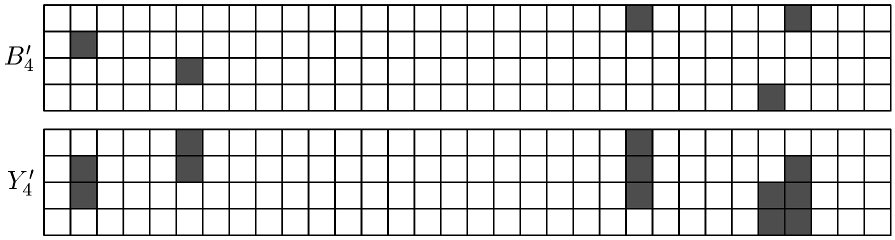
The second-round characteristic has probability 2^{-5}:



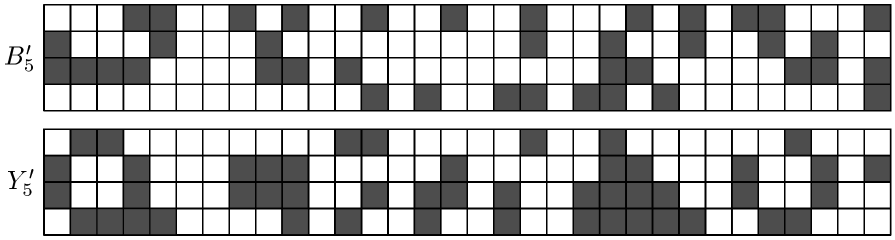
The third-round characteristic has probability 2^{-3}:



The fourth-round characteristic has probability 2^{-10}.



The fifth-round characteristic has probability 2^{-49}.



A.2 Boomerang Characteristic

The following is an example of a three-round characteristic with probability $p = 2^{-16}$. We used this characteristic in Section 4. This characteristic passes between rounds $i = 5 \bmod 8$ and $i = 7 \bmod 8$.

The fifth-round characteristic has probability 2^{-10}:

B'_5												■																				
					■	■						■																				
								■																								
					■	■		■																								

Y'_5																																
					■	■																										
								■																								
					■							■																				

The sixth-round characteristic has probability 2^{-2}:

B'_6																																
					■																											
					■																											

Y'_6																																
					■																											

The seventh-round characteristic has probability 2^{-4}:

B'_7																															■	
				■																												

Y'_7																																
																															■	
				■																											■	
				■																												

If we apply the linear transformation L to Y'_7, we get:

B'_8																■					■	■		■	■		■					
																											■			■		■
	■		■	■		■	■				■																			■	■	
																					■					■			■			

Session 5:

"Cryptographic Analysis and Properties"

(II)

Attacking Seven Rounds of Rijndael under 192-bit and 256-bit Keys

Stefan Lucks*

Theoretische Informatik
University of Mannheim, 68131 Mannheim, Germany
lucks@th.informatik.uni-mannheim.de

Abstract. The authors of Rijndael [3] describe the "Square attack" as the best known attack against the block cipher Rijndael. If the key size is 128 bit, the attack is faster than exhaustive search for up to six rounds. We extend the Square attack on Rijndael variants with larger keys of 192 bit and 256 bit. Our attacks exploit minor weaknesses of the Rijndael key schedule and are faster than exhaustive search for up to seven rounds of Rijndael.

1 Introduction

The block cipher Rijndael [3] has been proposed as an AES candidate and was selected for the secound round. It is a member of a fast-growing family of Square-like ciphers [2–4, 6, 7].

Rijndael allows both a variable block length of $M * 32$ bit with $M \in \{4, 6, 8\}$ and a variable key length of $N * 32$ bit, N an integer. In the context of this paper we concentrate on $M = 4$, i.e., on a block length of 128 bit, and on $N \in \{4, 6, 8\}$, i.e., on key sizes of 128, 192, and 256 bit. We abridge these variants by RD-128, RD-192 and RD-256. The number R of rounds is specified to be $R = 10$ for RD-128, $R = 12$ for RD-192, and $R = 14$ for RD-256. In the context of this paper, we consider reduced-round versions with $R \leq 7$.

The authors of Square [2] described the "Square attack", a dedicated attack exploiting the byte-oriented structure of Square. The attack works for Square reduced to six rounds and is applicable to Rijndael and other Square-like ciphers as well [3, 4, 1]. This paper deals with extensions of the Square-attack for RD-192 and RD-256.

In Section 2, we shortly describe Rijndael, leaving out many details and pointing out some properties relevant for our analysis. Section 3 deals with the Square attack for up to six rounds of Rijndael, originating from

* Supported by DFG grant Kr 1521/3-1.

[2, 3]. In Sections 4–6 we describe attacks for seven rounds of Rijndael. The attack in Section 4 and its analysis is valid for all versions of Rijndael, while the attacks in Section 5 and Section 6 are dedicatedly for Rijndael-256 and Rijndael-192, exploiting minor weaknesses of the Rijndael key schedule. We give final comments and conclude in Section 7.

2 A Description of Rijndael

Rijndael is a byte-oriented iterated block cipher. The plaintext (a 128-bit value) is used as initial state, the state undergoes a couple of key-dependent transformations, and the final state is taken as the ciphertext. A state $A \in \{0,1\}^{128}$ is regarded as a 4×4 matrix $(A_{i,j})$, $i, j \in \{0,1,2,3\}$ of bytes (see Figure 1). The four columns of A are $A_i = (A_{0,j}, A_{1,j}, A_{2,j}, A_{3,j})$.

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Fig. 1. The index positions (i, j) for a 4*4 matrix of bytes.

Given the initial state, R rounds of transformations are applied. Each round can be divided into several elementary transformations.

By the *key schedule*, the key, a $(N * 32)$-bit value with $N \in \{4, 6, 8\}$, is expanded into an array $W[\cdot]$ of $4(R+1)$ 32-bit words $W[0], \ldots, W[4(R+1)-1]$. Four such words $W[4r+j]$ with $j \in \{0,1,2,3\}$ together are used as r-th "round key" K^r, with $r \in \{0, \ldots, R\}$. Like the state, we regard a round key K^r as a $4 * 4$ matrix of bytes $K^r_{i,j}$ with four columns $K^r_j = W[4r+j]$ for $j \in \{0,1,2,3\}$.

2.1 The Elementary Transformations of Rijndael

Rijndael uses four elementary operations to transform a state $A = (A_{i,j})$ into a new state $B = (B_{i,j})$, see also Figure 2:

1. The *byte substitution* (BS): $B_{i,j} := S(A_{i,j})$ for $i, j \in \{0,1,2,3\}$. Here, S denotes a permutation over $\{0,1\}^8$, i.e., S^{-1} is defined with $A_{i,j} = S^{-1}(B_{i,j})$.
2. The *shift row* operation (SR), a cyclic shift of bytes: $B_{i,j} := A_{i,(j+i) \bmod 4}$.
3. The *mix column* transformation (MC). Each column A_i of state A is transformed via a linear transformation μ over $\{0,1\}^{32}$, i.e. $B_i := \mu(A_i)$ for $i \in \{0,1,2,3\}$. Also, μ is invertible.
   An input $X \in \{0,1\}^{32}$ for μ can be seen as a vector $X = (X_0, X_1, X_2, X_3)$ of four bytes. Consider $X' = (X'_0, X'_1, X'_2, X'_3)$ to be different from X in exactly k bytes ($1 \le k \le 4$), i.e.

   $$k \;=\; \left|\{\; i \in \{0,1,2,3\} \mid X_i \neq X'_i \;\}\right|.$$

   Then $Y = \mu(X)$ and $Y' = \mu(X')$ are different in at least $5-k$ of their four bytes. The same property holds for the inverse μ^{-1} of μ.
4. The *key addition* (KA). The r-th round key $K^r = (K^r_{i,j})$ is added to the state A by bit-wise XOR: $B_{i,j} := A_{i,j} \oplus K^r_{i,j}$.

Note that all elementary transformations of Rijndael are invertible.

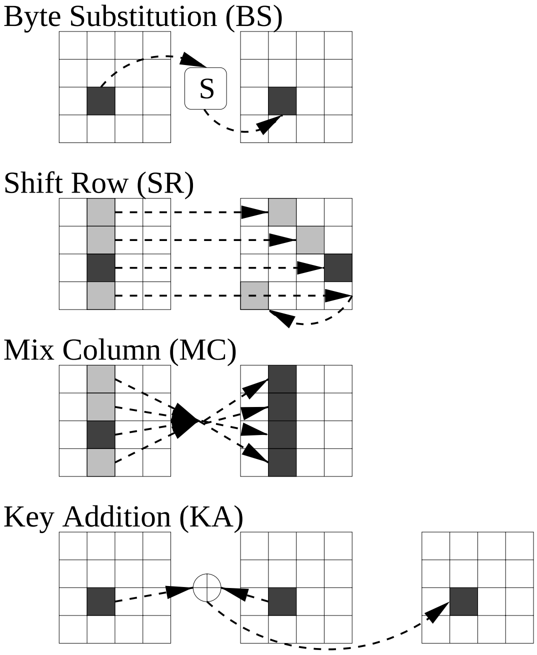


Fig. 2. The four elementary transformations of Rijndael.

2.2 The Rijndael Round Transformation

For $r \in \{0, \ldots, R\}$, the round key K^r consists of the expanded key words $W[4r]$, ..., $W[4r+3]$. The structure of Rijndael is defined as follows[1]:

```
1. $S :=$ plaintext;
2. KA ($S$, $K^0$); (∗ add round key 0 before the first round ∗)
3. for $r := 1$ to $R$ do: (∗ run through round 1, 2, ..., $R$ ∗)
    4. $S :=$ BS($S$); (∗ byte substitution ∗)
    5. $S :=$ SR($S$); (∗ shift row ∗)
    6. $S :=$ MC($S$); (∗ mix column ∗)
    7. $S :=$ KA($S$, $K^r$); (∗ add round key $r$ ∗)
8. ciphertext $:= S$.
```

Steps 4–7 are the "standard representation" of the Rijndael round structure. The implementor of Rijndael has a great degree of freedom to change the order the elementary operations are done – without changing the behavior of the cipher. (We refer the reader to the description of the "algebraic properties" and the "equivalent inverse cipher structure" for details [3, Section 5.3].) We describe one alternative representation of the round structure. As an "alias" for the r-th round key K^r we use the value

$$L^r = \mathrm{SR}^{-1}(\mathrm{MC}^{-1}(K^r)). \tag{1}$$

Accordingly, we distinguish between the "L-representation" L^r of a round key and its "K-representation". Knowing L^r is equivalent to knowing K^r, and knowing a column K^r_j of K^r is equivalent to knowing four bytes of L^r, see Table 1.

known column of K^r	known bytes $L^r_{i,j}$ of L^r
K^r_0	$(i,j) \in \{(0,0),(1,3),(2,2),(3,1)\}$
K^r_1	$(i,j) \in \{(0,1),(1,0),(2,3),(3,2)\}$
K^r_2	$(i,j) \in \{(0,2),(1,1),(2,0),(3,3)\}$
K^r_3	$(i,j) \in \{(0,3),(1,2),(2,1),(3,0)\}$

Table 1. Known columns of a key in K-representation and the corresponding known key bytes in L-representation.

[1] Actually, the authors of Rijndael [3] specify an exception: in the last round, the MC-operation is left out. As was stressed in [3], this modification does not strengthen or weaken the cipher. In the current paper, we assume for simplicity that the last round behaves exactly like the other rounds.

The following describes a functionally equivalent round structure for Rijndael, see also Figure 3.

4. $S :=$ BS(S); (∗ byte substitution ∗)
5. $S :=$ KA(S, L^r); (∗ add round key, given in L- representation ∗)
6. $S :=$ SR(S); (∗ shift row ∗)
7. $S :=$ MC(S); (∗ mix column ∗)

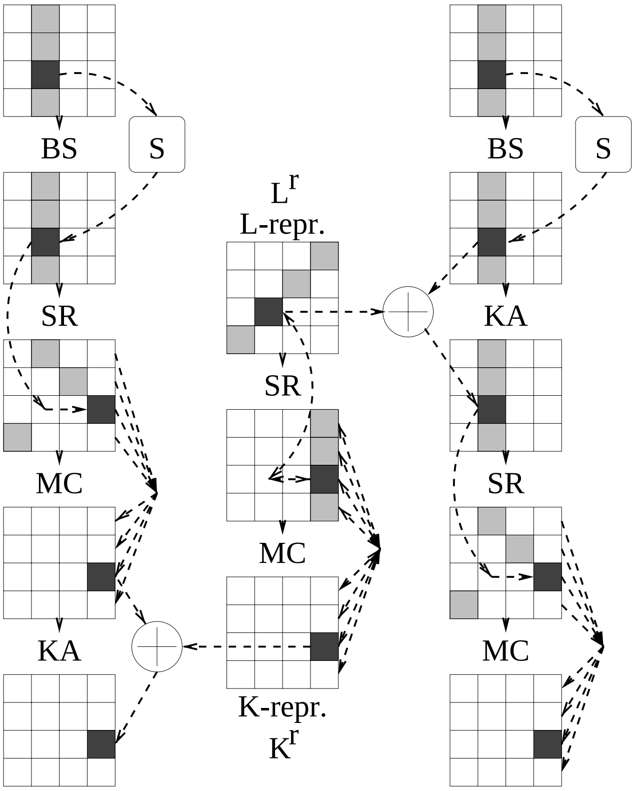


Fig. 3. The Structure of a Rijndael round.
Left: The standard representation of the Rijndael round transformation
Middle: The round key – changing between K-representation and L-representation
Right: The alternative representation of the Rijndael round transformation

2.3 The Rijndael Key Schedule

The key schedule is used to generate an expanded key from a short (128–256 bit) "cipher key". We describe the key-schedule using word-wise operations (where a word is a 32-bit quantity), instead of byte-wise ones. The cipher key consists of N 32-bit words, the expanded key of $4*(R+1)$ such words $W[\cdot]$. The first N words $W[0], \ldots, W[N-1]$ are directly initialised by the N words of the cipher key.

For $k \in \{1, 2, \ldots\}$, $\mathrm{const}(k)$ denotes fixed constants, and $f, g : \{0,1\}^{32} \rightarrow \{0,1\}^{32}$ are nonlinear permutations.[2] For $i \in \{N, \ldots, 4*(R+1)-1\}$ the words $W[i]$ are defined recursively:

$$\begin{array}{l} \text{If } (i \bmod N) = 0 \\ \quad \text{then } W[i] := W[i-N] \oplus f(W[i-1]) \oplus \mathrm{const}(i\,\mathrm{div} N) \\ \quad \text{else if } ((N > 6) \text{ and } (i \bmod N) = 4) \\ \qquad\quad \text{then } W[i] := W[i-N] \oplus g(W[i-1]) \\ \qquad\quad \text{else } W[i] := W[i-N] \oplus W[i-1]. \end{array} \tag{2}$$

Note that two words $W[i-1]$ and $W[i-N]$ suffice to compute the word $W[i]$. Similarly, we can go backwards: Given two words $W[i]$ and $W[i-1]$, we can compute $W[i-N]$. (This will be useful for our attacks below.) Hence, any N consecutive words $W[k], \ldots, W[k+N-1]$ of the expanded key suffice to efficiently generate the complete expanded key and thus to completely break Rijndael.

3 The Square Attack for Rijndael

In this section we describe the dedicated Square-Attack for Rijndael. More details can be found in [2, 3]. We start with a simple attack on four rounds and extend the simple attack by an additional round at the beginning and another one at the end. This leads to the "Square-6" attack for six rounds of Rijndael. Analysing the performance of our attacks with respect to RD-192 and RD-256 is delayed until the end of this section.

3.1 Attacking Four Rounds – the Simple Attack

To describe the attack we need the notion of a "Λ-set", i.e., a set of 2^8 states that are all different in some of their $4*4$ bytes (the "active" bytes), and all equal in the other ("passive") bytes. In other words, for two distinct states A and B in a Λ-set we always have

$$\begin{array}{ll} A_{i,j} \neq A_{i,j} & \text{if the byte at position } (i,j) \text{ is active, and} \\ A_{i,j} = B_{i,j} & \text{else, i.e., if the byte at } (i,j) \text{ is passive.} \end{array}$$

[2] We omit the definition of f and g, but we point out that the four functions f, g, f^{-1} and g^{-1} are fixed in the definition of Rijndael and can be computed efficiently.

A Λ-set with exactly k active bytes is a "Λ^k-set".

The adversary chooses one Λ^1-set P_0 of states (plaintexts). By P_i we denote the sets of 2^8 states which are the output of round i. P_1 is a Λ^4-set, all four active bytes in the the same column. P_2 is a Λ^{16}-set. P_3 is unlikely to be a Λ-set. But, as explained in [2,3], all the bytes of S_3 are "balanced", i.e., the following property holds:

$$\text{For all } (i,j) \in \{0,1,2,3\}^2 : \bigoplus_{A \in P_3} A_{i,j} = 0. \tag{3}$$

Recall that we consider a four-round attack, i.e., P_4 is the set of 2^8 ciphertexts the adversary learns. It is unlikely that the bytes of P_4 are balanced, but the balancedness of the bytes of P_3 can be exploited to find the fourth round key K^4. Let L^4 be the L-representation of K^4, cf. Equation (1). The attack defines a set Q_4 "in between"[3] P_3 and P_4:

```
1. For $X \in P_4$:
       $Y := \mathrm{MC}^{-1}(X)$;
       $Z := \mathrm{SR}^{-1}(Y)$.
   Denote the set of $2^8$ states $Z$ by $Q_4$.
2. For all $(i,j) \in \{0,1,2,3\}^2$:
       for $a \in \{0,1\}^8$:
           $b(a) := \bigoplus_{Z \in Q_4} S^{-1}(Z_{i,j} \oplus a)$;
           if $b(a) \neq 0$ then conclude $L^4_{i,j} \neq a$.
```

In short, we invert round four step by step: invert the mix column operation, invert the shift row operation, add (a possible choice for) the key byte $L^4_{i,j}$ and invert the byte substitution. If the guess $a \in \{0,1\}^8$ for the key $L^4_{i,j}$ is correct, the set of 2^8 bytes $S^{-1}(Z_{i,j} \oplus a)$ is balanced, i.e., $b(a) = 0$. But if our guess a' for $L^4_{i,j}$ is wrong, we estimate $b(a') = 0$ to hold with only a probability of 2^{-8}. Thus, on the average two candidates for for each byte $L^4_{i,j}$ are left – the correct byte and a wrong one. We can easily reconstruct an expected number of less than 2^{16} candidates for L^4.

Each candidate corresponds with a unique choice for the 128-bit cipher key of RD-128. To find the cipher key, we may either choose a second Λ^1-set of plaintexts, or just use exhaustive search over all key candidates, using the same 2^8 known pairs of plaintext and ciphertext as before. With overwhelming probability, either approach uniquely determines the

[3] In general, we regard Q_5 to be a set of states "in between" P_{r-1} and P_r. Note that converting a state in P_r into its counterpart in P_4 does not depend on the key and can be just like converting a round key from its K-representation into its L-representation, similarly to Equation (1).

cipher key, using few memory and an amount of work determined by step 2, i.e., about 2^{20} byte-wise XOR-operations. Note that the first approach needs twice as many chosen plaintexts as the second one.

3.2 An Extension at the End

As suggested in [2, 3], the above basic attack can be extended by one additional round at the beginning and another round at the end. We start with extending the additional round at the end.

Let P_0 be chosen as above. By P_5, we denote the set of 2^8 outputs of round 5. Similar to Q_4 above, the adversary can find Q_5 by applying MC^{-1} and applying SR^{-1}. Given the set Q_5, we can compute P_3 by inverting $1\frac{1}{2}$ rounds of Rijndael.

If the set P_5 (or Q_5) is fixed, the bytes of P_3 at position, say, $(0, 1)$ only depend on $L^5_{0,1}$, $L^5_{1,1}$, $L^5_{2,1}$, $L^5_{3,1}$, and on $L^4_{0,1}$, see Figure 4.

Similar to the four-round attack, we may guess such a five-tuple of key bytes and compute the corresponding bytes of P_3. If these aren't balanced, we reject the corresponding key bytes. We expect one out of 2^8 incorrect five-tuples to be *not* rejected. With five Λ-sets of plaintexts, i.e., $5 * 2^8$ chosen plaintexts, the the cipher key can easily be found via exhaustive search. (A more diligent treatment would allow us to reduce the number of chosen plaintexts for this attack, but without much effect on the required number of chosen plaintexts for the six-round attack below.)

To measure the running time of our attacks, we use the notion of a "basic operation". Given a column Y_j of bytes of a state Y in Q_r, the key column L^r_j and another key byte $L^{r-1}_{k,j}$ with the row index k as the input, we compute the byte $X_{k,j}$ of a state X in P_{r-2}, using $V = (V_0, V_1, V_2, V_3)$ and $W = (W_0, W_1, W_2, W_3)$ as intermediate values and define the basic operation $X_{k,j} = \mathrm{BO}(Y_j, k, L^r_j, L^{r-1}_{k,j})$ as follows:

```
1. For $i := 0$ to 3: $V_i := S^{-1}(Y_{i,j} \oplus L^r_{i,j})$.
2. $W := \mu^{-1}(V)$.
3. $X_{k,j} := S^{-1}(W_{k,j} \oplus L^{r-1}_{k,j})$.
```

In short: one basic operation requires 5 byte-wise XORs, 5 evaluations of S^{-1}, and one evaluation of μ^{-1}.

To check the correctness of a quintuple of bytes, we have to do 2^8 basic operations and to XOR the results for a balance-check by verifying Equation (3). We do this for every quintuple of bytes. Thus, the five-round attack takes the time of about 2^{48} basic operations.

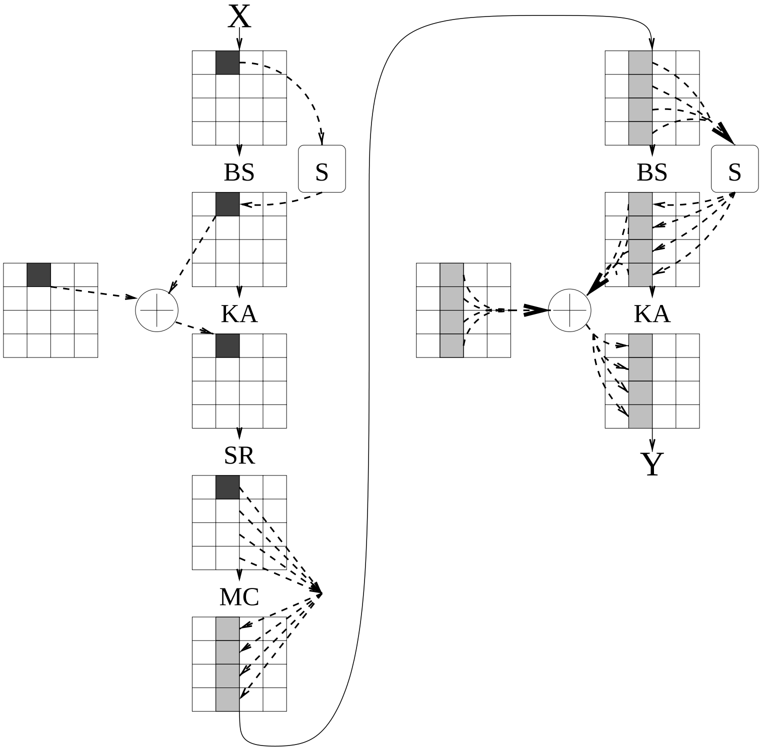


Fig. 4. $1\frac{1}{2}$ rounds of Rijndael: Given one column Y_j of the output state Y and five corresponding key bytes, one can find one byte $X_{k,j}$ of the input X by inverting these $1\frac{1}{2}$ rounds of Rijndael; we write $X_{k,j} = \mathrm{BO}(Y_j, k, \textit{key bytes})$ and consider this a "basic operation" for our attacks.

3.3 Attacking Six Rounds – the Square-6 attack

Now we extend the above attack by an additional "round 0". We denote this attack the "Square-6 attack".

Let P_0 be a Λ^1-set, as before. By doing one additional round of *de*cryption, we get a Λ^4-set P_{-1}. The active bytes of P_{-1} are at positions determined by the positions of the active P_0-byte. E.g., if the P_0-byte at position $(0, 0)$ is the active one, the P_{-1}-bytes at positions $(0, 0)$, $(1, 3)$, $(2, 2)$, and $(3, 1)$ are active.

The idea is to arbitrarily fix the plaintext bytes at the passive positions and to choose 2^{32} plaintexts varying at the active positions. If the four corresponding bytes of the round key K^{-1} for round 0 are known (e.g. for the active P_0-byte at position $(0, 0)$: if $K^{-1}_{0,0}$, $K^{-1}_{1,3}$, $K^{-1}_{2,2}$ and

$K_{3,1}^{-1}$ are known), the adversary can easily determine many 2^8-sets P_{-1} of plaintexts, such that the sets P_0 are Λ^1-sets.

The adversary accordingly chooses 2^{32} plaintexts and, for all 2^{32} relevant key bytes, runs the five-round attack described above. This is 2^{32} times slower than the five-round attack itself, i.e. takes about 2^{80} basic operations. The memory requirement for this attack is dominated by the need to store 2^{32} ciphertexts.

3.4 Considering RD-192 and RD-256

Note that the six-round Square-6 attack and the five-round attack allow us to finds two round keys K^4 and K^5 at the same time. (The attacker chooses a five-tuple of key bytes, one byte from K^4 and four from K^5, and probabilistically verify if that choice is correct.) Once the attacker knows two consecutive round keys, i.e. eight consecutive words from the expanded key, the attacker can easily run the key schedule backwards to find the cipher key. In other words, the performance of the Square-6 attack does not depend on which flavor of Rijndael we attack, RD-128, RD-192, or RD-256. We call such an attack a "generic" attack.

The simple four-round attack only provides the attacker with the round key K^4. But finding the round key K^3 is easy, since the set P_2 of states is a Λ^{16}-set.

4 A Generic Attack for Seven Rounds of Rijndael

It is easy to extend the Square-6 attack to seven rounds of Rijndael:

1. Choose 2^{32} input plaintexts for the Square-6 attack and ask for the corresponding ciphertexts.
2. For all $K^7 \in \{0,1\}^{128}$:
   - **3.** Last-round-decrypt the 2^{32} ciphertexts under K^7.
   - **4.** Run the Square-6 attack for the results, to get the round keys K^6 and K^5.
   - **5.** Given the round keys K^5, K^6 and K^7, we have more than sufficient key material to recover the complete extended key and to check it for correctness.

The seven-round attack requires the same amount of chosen plaintexts and memory as the Square-6 attack. The running time increases by a factor of 2^{128}, i.e. to the equivalent of

$$2^{80} * 2^{128} = 2^{208} \text{ basic operations.}$$

Even though the attack is generic, it is pointless for attacking either RD-128 or RD-192 – exhaustive key search is much faster for these variants of Rijndael.

5 Attacking Seven Rounds of RD-256

For RD-256, the above generic seven-round attack improves on exhaustive search. But, as shown here, the RD-256 key schedule allows us to accelerate the attack by a factor of 2^8.

Note that if we know (or have chosen) K^7, we know the expanded key words $W[28]$, $W[29]$, $W[30]$, and $W[31]$. By Formula (2), we get

$$\begin{aligned} W[21] &= W[28] \oplus W[29], \\ W[22] &= W[29] \oplus W[30], \text{ and} \\ W[23] &= W[30] \oplus W[31]. \end{aligned}$$

Hence, we know know three columns of K^5, including e.g. K_1^5. As explained in Section 2.3, this implies knowing 12 bytes of L^5, including e.g. $L_{0,1}^5$. To test the bytes of 256-set P_4 at position $(0,1)$, we need the bytes in column 1 from Q_6, the corresponding key column L_1^6 from L^6 and the key byte $L_{0,1}^5$ from L^5 (cf. Figure 4 at page 9). We attack seven rounds of RD-256 by the following algorithm:

```
1. Choose $2^{32}$ distinct input plaintexts, varying at the byte positions
   $(0,0)$, $(1,2)$, $(2,2)$, and $(3,1)$ and constant at the other byte positions.
   Ask for the corresponding ciphertexts.
2. For all $2^{32}$ combinations of $K_{0,0}^0$, $K_{1,3}^0$, $K_{2,2}^0$, $K_{3,1}^0$:
   3. Fix 32 distinct sets $P_0[i]$ of plaintexts ($i \in \{0, \dots, 31\}$) with $|P_0[i]| =$
      $2^8$, such that the corresponding $P_1[i]$ are $\Lambda^1$-sets.
   4. For all $2^{128}$ round keys $K^7$:
      5. Decipher the 32 sets of ciphertexts $P_7[i]$ to get $P_6[i]$ and $Q_6[i]$.
      6. Compute $L_{0,1}^5$.
      7. For all $2^{32}$ combinations of $L_1^6 = (L_{0,1}^6, L_{1,1}^6, L_{2,1}^6, L_{3,1}^6)$:
         8. $i := 0$; reject := false;
         9. while $i \leq 31$ and reject=false:
            begin
               10. Compute
                   $$b[i] := \bigoplus_{A \in Q_6[i]} \mathrm{BO}(A_1, 1, L_1^6, L_{0,1}^5).$$
      **11.** If $b[i] = 0$ then i := i+1
                            else reject := true.
    end (* while *).
  **12.** If reject=false then stop (* and accept key bytes *).
```

The above algorithm exhaustively searches a subspace of size 2^{192} of the full key space. When all 24 key bytes are correct, step 11 always executes then-clause and increments the counter i. After 32 such iterations, the algorithm stops in step 12.

If any of the 24 byte key bytes is wrong, we execute the then-clause only with a probability of 2^{-8}. Since the counter i runs from 0 to 31, the probability for a wrong 24-tuple of key bytes to be accepted is below $2^{-8*32} = 2^{-256}$. There are only 2^{192} such tuples of key bytes, thus the probability to accept any wrong 24-tuple is less than $2^{32} * 2^{128} * 2^{32} * 2^{-256} \leq 2^{-64}$, i.e. negligible.

When stopping, the algorithm accepts K^7 and four bytes of K^6 (or L^6). By exhaustive search, it is easy to find the other 12 bytes of K^6. Having done that, the key schedule allows to find the full expanded key.

For the attack, 2^{32} chosen plaintexts suffice, and the required storage space is dominated by the need to store the corresponding 2^{32} ciphertexts.

What about the running time? The loop in step 2 is iterated 2^{32} times, step 4 takes 2^{128} iterations, and the loop in step 7 is iterated 2^{32} times. On the average, the while-loop is iterated $1 + 2^{-8} + 2^{-16} + \ldots$ times, i.e., about once. Step 12 needs 2^8 basic operations. This makes about

$$2^{32} * 2^{128} * 2^{32} * 1 * 2^8 = 2^{200} \text{ basic operations.}$$

6 Attacking Seven Rounds of RD-192

In the case of RD-192, accelerating the generic attack by a factor of 2^8, as in the case of RD-256, would still not suffice to outperform exhaustive search. Fortunately (for the cryptanalyst), the RD-192 key schedule allows an acceleration by a factor of 2^{24}, compared to the generic attack,

The columns of K^7 are the words $W[28]$, $W[29]$, $W[30]$, and $W[31]$ of the expanded key. These four words allow us to compute three more words – in the case of RD-192, these are $W[23]$, $W[24]$, and $W[25]$, cf. Section 2.3. Two of these words are columns of the round key K^6, while the third word is a column of K^5: $W[24] = K^6_0$, $W[25] = K^6_1$, and $W[23] = K^5_3$.

From $W[23]$, $W[24]$, and $W[25]$, we can compute three useful key bytes for the attack, for example $L^5_{0,3}$, $L^6_{1,3}$, and $L^6_{2,3}$, cf. Table 1 on page 4. The two remaining key bytes (in our example $L^6_{0,3}$ and $L^6_{3,3}$) still have to be found:

```
1. Choose $2^{32}$ distinct input plaintexts, varying at the byte positions
   $(0,0)$, $(1,2)$, $(2,2)$, and $(3,1)$ and constant at the other byte positions.
   Ask for the corresponding ciphertexts.
2. For all $2^{32}$ combinations of $K^0_{0,0}$, $K^0_{1,3}$, $K^0_{2,2}$, $K^0_{3,1}$:
   3. Fix 32 distinct sets $P_0[i]$ of plaintexts ($i \in \{0, \ldots, 31\}$) with $|P_0[i]| =$
      $2^8$, such that the corresponding $P_1[i]$ are $\Lambda^1$-sets.
   4. For all $2^{128}$ round keys $K^7$:
      5. Decipher the 32 sets of ciphertexts $P_7[i]$ to get $P_6[i]$ and $Q_6[i]$.
      6. Compute $L^5_{0,3}$, $L^6_{1,3}$, and $L^6_{2,3}$.
      7. For all $2^{16}$ combinatios of $L^6_{0,3}$ and $L^6_{3,3}$:
         8. $i := 0$; reject := false;
         9. while $i \leq 31$ and reject=false:
            begin
               10. Compute

                   $b[i] := \bigoplus_{A \in Q_5[i]} \mathrm{BO}(A_3, 3, L^6_1, L^5_{0,1}).$

               11. If $b[i] = 0$ then i := i+1
                                 else reject := true.
            end (* while *).
         12. If reject=false then stop (* and accept key bytes *).
```

The analysis of the attack is essentially the same as its counterpart for RD-256. The only difference is that the loop in step 7 is iterated 2^{16} times instead of 2^{32}. So the attack needs the time of about

$$2^{32} * 2^{128} * 2^{16} * 1 * 2^8 = 2^{184} \text{ basic operations.}$$

7 Final Comments, Summary, and Conclusion

In [3], the authors of Rijndael described the Square-6 attack for RD-128. Extensions of this attack for RD-192 and RD-256 were missing, though. The target of the current paper is to close this gap.

The attacks described in this paper are highly impractical. Considering even such certificational attacks as ours is good scientific practice. And the design of Rijndael was determined "by looking at the maximum number of rounds for which shortcut attacks have been found" [3, Chapter 7.6], allowing an additional margin of security. Any attack which is faster than exhaustive search counts as "shortcut attack".

Attack	target	# Rounds	# Chosen Plaintexts	Time [# basic operations]	Memory [# Ciphertexts]
simple Square	generic	4	2^8	small	small
ext. at the end	generic	5	$5 * 2^8$	2^{48}	small
Square-6	generic	6	2^{32}	2^{80}	2^{32}
7-round	generic	7	2^{32}	2^{208}	2^{32}
	RD-192	7	2^{32}	2^{184}	2^{32}
	RD-256	7	2^{32}	2^{200}	2^{32}

Table 2. Summary of Results.

Table 2 summarises how the different attacks perform. The results for 4–6 rounds of Rijndael originate from [3]. Note that [3] counted the number of "cipher executions" to measure the running time.

In [5], Fergusen and others describe improved attacks on Rijndael. A preliminary version of [5] has been sent to the current author by one of the authors of [5], and the following remarks are base on that version. [5] describes some weaknesses of the Rijndael key schedule, but does not exploit these for actual attacks. The attacks in [5] are mainly based on improvements of the Square-6 attack, using the "partial sums". E.g., an attack on seven rounds of Rijndael is proposed, which requires 2^{32} chosen plaintexts and the time equivalent to 2^{170} trial encryptions. Using the observations made in the current paper, this attack can be improved by a factor of 2^{16}, i.e., only needs the equivalent of 2^{156} trial encryptions, instead of 2^{170}.

Our results exhibit a weakness in the Rijndael key schedule. If, e.g., the words $W[\cdot]$ of the expanded key were generated pseudorandomly using a cryptographically secure pseudorandom bit generator, dedicated attacks could not be more efficient than their generic counterparts.

This does not indicate the necessity to modify the Rijndael key schedule, though. The improvements on the generic case are quite small. If we concentrate on counting the number of rounds for which shortcut attacks exist, the cryptanalytic gain of this paper is one round for RD-192, not more. The authors of Rijndael seem to have anticipated such cryptanalytic results by specifying a high security margin for the number of rounds (two additional rounds for RD-192, compared to RD-128 with its ten rounds).

References

1. C. D'Halluin, G. Bijnens, V. Rijmen, B., Preneel: "Attack on six round of Crypton", Fast Software Encryption 1999, Springer LNCS 1636, pp. 46–59.
2. J. Daemen, L. Knudsen, V. Rijmen: "The block cipher Square", Fast Software Encryption 1997, Springer LNCS 1267, pp. 149–165.
3. J. Daemen, V. Rijmen: "AES proposal: Rijndael" (2nd version), AES submission.
4. J. Daemen, V. Rijmen: "The block cipher BKSQ", Cardis 1998, Springer LNCS, to appear.
5. N. Ferguson, J. Kelsey, B. Schneier, M. Stay, D. Wagner, D. Whiting: "Improved Cryptanalysis of Rijndael", Fast Software Encryption 2000, Springer LNCS, to appear.
6. C. H. Lim: "Crypton: a new 128-bit block cipher", AES submission.
7. C. H. Lim: "A revised version of Crypton – Crypton V 1.0 – ", Fast Software Encryption 1999, Springer LNCS 1636, pp. 31–45.

A collision attack on 7 rounds of Rijndael

Henri Gilbert and Marine Minier

France Télécom R & D
38-40, rue du Général Leclerc
92794 Issy les Moulineaux Cedex 9 - France
email : henri.gilbert@cnet.francetelecom.fr

Abstract

Rijndael is one of the five candidate blockciphers selected by NIST for the final phase of the AES selection process. The best attack of Rijndael so far is due to the algorithm designers ; this attack is based upon the existence of an efficient distinguisher between 3 Rijndael inner rounds and a random permutation, and it is limited to 6 rounds for each of the three possible values of the keysize parameter (128 bits, 196 bits and 256 bits). In this paper, we construct an efficient distinguisher between 4 inner rounds of Rijndael and a random permutation of the blocks space, by exploiting the existence of collisions between some partial functions induced by the cipher. We present an attack based upon this 4-rounds distinguisher that requires 2^{32} chosen plaintexts and is applicable to up to 7-rounds for the 196 keybits and 256 keybits version of Rijndael.
Since the minimal number of rounds in the Rijndael parameter settings proposed for AES is 10, our attack does not endanger the security of the cipher, indicate any flaw in the design or prove any inadequacy in selection of number of rounds. The only claim we make is that our results represent improvements of the previously known cryptanalytic results on Rijndael.

1 Introduction

Rijndael [DaRi98], a blockcipher designed by Vincent Rijmen and Joan Daemen, is one of the 5 finalists selected by NIST in the Advanced Encryption Standard competition [AES99]. It is a variant of the Square blockcipher, due to the same authors [DaKnRi97]. It has a variable block length b and a variable key length k, which can be set to 128, 192 or 256 bits. The recommended nr number of rounds is determined by b and k, and varies between 10 and 14. In the sequel we will sometimes use the notation Rijndael/b/k/nr to refer to the Rijndael variant determined by a particular choice of the b, k and nr parameters.

The best Rijndael attack published so far is due to the algorithm designers [DaRi98]. It is a variant of a the ”Square” attack, and exploits the byte-oriented structure of Rijndael [DaKnRi97]. This attack is based upon an efficient distinguisher between 3 Rijndael inner rounds and a random permutation. It is stated in [DaRi98] that ”for the different block lengths of Rijndael no extensions to 7 rounds faster than exhaustive search have been found”.
In this paper we describe an efficient distinguisher between 4 Rijndael inner rounds and a random permutation, and we present resulting 7-rounds attacks of Rijndael/b=128 which are substantially faster than an exhaustive key search for the $k = 196$ bits and $k = 256$ bits versions and marginally faster than an exhaustive key search for the $k = 128$ bits version.

This paper is organised as follows. Section 2 provides an outline of the cipher. Section 3 investigates partial functions induced by the cipher and the existence of collisions between such partial functions, and describes a resulting distinguisher for 4 inner rounds. Section 4 presents 7-rounds attacks based on the 4-rounds distinguisher of Section 3. Section 5 concludes the paper.

2 An outline of Rijndael/$b = 128$

In this Section we briefly described the Rijndael algorithm. We restrict our description to the b=128 bits blocksize and will consider no other blocksize in the rest of this paper.
Rijndael/$b/k/nr$ consists of a key schedule and an iterated encryption function with nr rounds. The key schedule derives $nr + 1$ 128-bit round keys K_0 to K_{nr} from the $k = 128, 196$ or 256 bits long Rijndael key K. Since attacks presented in the sequel do not use the details of the dependence between round keys, we do not provide a description of the key schedule.
The Rijndael encryption function is the composition of nr block transformations. The current 128-bit block value B is represented by a 4 × 4 matrix :

$$B = \begin{array}{|c|c|c|c|}\hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline\end{array}$$

The definition of the round functions involves four elementary mappings :

- the σ=ByteSub byte substitution transforms each of the 16 input bytes under a fixed byte permutation P (the Rijndael S-box).
- the ρ=ShiftRow rows shift circularly shifts row i ($i = 0$ to 3) in the B matrix by i bytes to the right.
- the μ=MixColumn is a matrix multiplication by a fixed 4 × 4 matrix of non-zero GF(2^8) elements.
- the κ_r=KeyAddition is a bitwise addition with a 128-bit round key K_r.

The Rijndael cipher is composed by an initial round key addition κ_0, $nr-1$ inner rounds and a final transformation. The rth inner round ($1 \leq r \leq nr-1$) is defined as the $\kappa_r \circ \mu \circ \rho \circ \sigma$ function. The final transformation at the round nr is an inner round without MixColumn mapping : FinalRound $= \kappa_{nr} \circ \rho \circ \sigma$. We can thus summarise the cipher as follows:

```
$B$:=$\kappa_0(B)$;
For $r = 1$ to $nr-1$
    $B$:=InnerRound($B$);
FinalRound($B$);
```

Remarks :

- σ is the single non $GF(8)$-linear function of the whole cipher.
- The Rijndael S-box P is the composition of the multiplicative inverse function in GF(8) (NB : '00' is mapped into itself) and a fixed GF(2)-affine byte transformation. If the affine part of P was omitted, algebraic methods (e.g. interpolation attacks) could probably be considered for the cryptanalysis of Rijndael.
- The $\mu \circ \rho$ linear part of Rijndael appears to have been carefully designed. It achieves a full diffusion after 2 rounds, and the Maximum Distance Separability (MDS) property of μ prevents good differential or linear "characteristics" since it ensures that two consecutive rounds involve many active S-boxes.

3 Distinguishing 4 inner rounds of Rijndael/b=128 from a random permutation

3.1 Notation

Figure 1 represents 4 consecutive inner round functions of Rijndael associated with any 4 fixed unknown 128-round keys. Y, Z, R, S represent the input blocks of the 4 rounds and T represents the output of the 4th round. We introduce short notations for some particular bytes of Y, Z, R, S, T, which play a particular role in the sequel : $y = Y_{0,0}$, $z_0 = Z_{0,0}$, $z_1 = Z_{1,0}$, $z_2 = Z_{2,0}$, $z_3 = Z_{3,0}$, and so on. Finally we denote by c the ($c_0 = Y_{1,0}$, $c_1 = Y_{2,0}$, $c_2 = Y_{3,0}$) triplet of Y bytes.

Let us fix all the Y bytes but y to any 11-uple of constant values. So the c triplet is assumed to be equal to a constant $c = (c_0, c_1, c_2)$ triplet, and the 12 $Y_{i,j}$, i=1 to 3, j=0 to 3 are also assumed to be constant. The Z, R, S, T bytes z_0 to z_3, r_0 to r_3, s, and t_0 to t_3 introduced in Figure 1 can be seen as c-dependent functions of the y input byte. In the sequel we sometimes denote by $z_0^c[y]$ to $z_3^c[y]$, $r_0^c[y]$ to $r_3^c[y]$, $s^c[y]$, $t_0^c[y]$ to $t_3^c[y]$ the z_i, r_i, s, t_i byte value associated with a c constant and one $y \in 0..255$ value.

X				y	
c_0					Y
c_1					
c_2					

↓ (4 key-dependent bytes)

X				z_0	
X				z_1	Z
X				z_2	
X				z_3	

↓ (4 c-dep and k-dep bytes)

X				r_0	
			X	r_1	R
		X		r_2	
	X			r_3	

↓ (1 key-dependent byte)

X				s	
					S

↑

X				t_0	
X				t_1	T
X				t_2	
X				t_3	

Figure 1: 4 inner rounds of Rijndael

3.2 The 3-rounds distinguisher used in the Rijndael/b=128 designers' attack

The Rijndael designers' attack is based upon the observations that :
- bytes z_0 to z_3 are one to one functions of y and the other Z bytes are constant.
- bytes r_0 to r_3 are one to one functions of y (as well as the 12 other R bytes).
- s is the XOR of four one to one functions of y and thus $\sum_{y=0}^{255} s[y] = 0$.

Thus 3 consecutive inner rounds of Rijndael have the distinguishing property that if all Y bytes but y are fixed and y is taken equal to each of the 256 possible values, then the sum of the 256 resulting s values is equal to zero.

This leads to a 6-rounds attack (initial key addition followed by 5 inner rounds followed by final round). As a matter of fact an initial round (i.e. an initial key addition followed by 1 inner round) can be added on top, at the expense of testing assumptions on 4 key bytes of the initial key addition. Moreover, two additional rounds can be added at the end (namely one additional inner round followed by one final round), at the expense of testing assumptions on 4 final round key bytes. Combining both extensions provides an attack which requires 2^{32} plaintexts and has a complexity of 2^{72} encryptions.

3.3 A 4-rounds distinguisher for Rijndael/$b = 128$

We now analyse in detail the dependency of the byte oriented functions introduced in Section 3.1 in the c constant and the expanded key. We show that the $s^c[y]$ function is entirely determined by a surprisingly small number of unknown bytes, which either only depend upon the key or depend upon both the key and the c value, and that as a consequence there exist (c', c'') pairs of distinct c values such that the $s^{c'}[\cdot]$ and $s^{c''}[\cdot]$ partial functions collide, i.e. $s^{c'}[y] = s^{c''}[y]$ for $y = 0, 1, \cdots, 255$. This provides an efficient test for distinguishing 4 inner rounds of Rijndael from a random permutation.

The construction of the proposed distinguisher is based upon the following observations, which are illustrated in Figure 1.

Property 1 : At round 1, the $y \rightarrow z_0^c[y]$ one to one function is independent of the value of the c triplet and is entirely determined by one key byte. The same property holds for z_1, z_2, z_3. This is because at the output of the first round ShiftRow the c_0 to c_2 constants only affect columns 1 to 3 of the current block value, whereas the z_0 to z_3 bytes entirely depend upon column 0. For similar reasons, the other bytes of Z are independent of y : each of the bytes of column 1 (resp 2, resp 3) of Z is entirely determined by the c_0 (resp c_1, resp c_2) byte and one key-dependent byte.
More formally, there exist 16 MixColumn matrix coefficients $a_{i,j}$,i=0..3, j=0..3 and 16 key-dependent constants $b_{i,j}$, i=0..3, j=0..3 such that $z_i = a_{i,0}P(y) + b_{i,0}$, i=0..3 and $z_{i,j} = a_{i,0}P(c_{j-1}) + b_{i,j}$, i=1..3, j=0..3.

Property 2 : At round 2, each of the four bytes $r_i[y]$, $i = 0..3$ is a one to one function of $z_i[y]$, and the $r_i[y] \rightarrow z_i[y]$ is entirely determined by one single unknown constant byte that is entirely determined by c and the key.

More formally, there exist 16 MixColumn coefficients α_i, $i = 0..3$, β_i, $i = 0..3$ γ_i, $i = 0..3$ and δ_i, $i = 0..3$ and 4 key-dependent constants ϵ_i, $i = 0..3$ such that $r_i = \alpha_i \cdot P(z_{i,0}) + \beta_i \cdot P(z_{i,1}) + \gamma_i \cdot P(z_{i,2}) + \delta_i \cdot P(z_{i,3}) + \epsilon_i$, $i = 0..3$. The r_i bytes are thus related to c and y by the relations : $r_i = \alpha_i \cdot P(a_{i,0}P(y) + b_{i,0}) + \beta_i \cdot P(a_{i,1}P(c_0) + b_{i,1}) + \gamma_i \cdot P(a_{i,2}P(c_1) + b_{i,2}) + \delta_i \cdot P(a_{i,3}P(c_2) + b_{i,3}) + \epsilon_i$, $i = 0..3$.
Consequently, the $r_0[y]$ to $r_3[y]$ one to one functions of y are entirely determined by the 4 key-dependent constant unknown bytes $b_{i,0}$ introduced in property (1) and the 4 c- and k-dependent bytes $b_i = \beta_i \cdot P(a_{i,1}P(c_0) + b_{i,1}) + \gamma_i \cdot P(a_{i,2}P(c_1) + b_{i,2}) + \delta_i \cdot P(a_{i,3}P(c_2) + b_{i,3}) + \epsilon_i$, i= $0 \cdot\cdot 3$.

Property 3 : At round 3, the s byte can be expressed as a function of the r_0 to r_3 bytes and one c-independent and key-dependent unknown constant. Consequently, the $s^c[y]$ function is entirely determined by 4 key-dependent and c-dependent constants and 5 c-independent and key-dependent constants.

Property 4 : Let us consider the decryption of the fourth inner round : s can be expressed as $s = p^{-1}[(0E.t_0 + 0B.t_1 + 0D.t_2 + 09.t_3) + k_5]$ where p represents the single S-box. In other words $0E.t_0 + 0B.t_1 + 0D.t_2 + 09.t_3$ is a one to one function of s, and that function is entirely determined by one single key byte k_5. Thus $0E.t_0 + 0B.t_1 + 0D.t_2 + 09.t_3$ is a function of y that is entirely determined by 6 unknown bytes which only depend upon the key and by 4 additional unknown bytes which depend both upon c and the key.

The above properties provide an efficient 4-rounds distinguisher. We can restate property (3) in saying that the $s^c[y]$ function is entirely determined (in a key-dependent manner) by the 4 c-dependent bytes b_0 to b_3. Let us make the heuristic assumption that these 4 unknown c-dependent bytes behave as a random function of the c triplet of bytes. By the birthday paradox, given a C set of about 2^{16} c triplet values, there exist with a non negligible probability two distinct c' and c'' in C such that the $s^{c'}[y]$ and $s^{c''}[y]$ functions induced by c' and c'' are equal (i.e. in other words such that the $(s^{c'}[y])_{y=0..255}$ and $(s^{c''}[y])_{y=0..255}$ lists of 256 bytes are equal). Property (4) provides a method to test such a "collision", using the t_0 to t_3 output bytes of 4 inner rounds : c' and c'' collide if and only if $\forall y \in [0,...,255]$, $0E.t_0^{c'} + 0B.t_1^{c'} + 0D.t_2^{c'} + 09.t_3^{c'} = 0E.t_0^{c''} + 0B.t_1^{c''} + 0D.t_2^{c''} + 09.t_3^{c''}$. Note that it is sufficient to test the above equality on a limited number of y values (say 16 for instance) to know with a quite negligible "false alarms" probability whether the $s^{c'}[y]$ and $s^{c''}[y]$ functions collide.

We performed some computer experiments which confirmed the existence, for arbitrarily chosen key values, of (c', c'') pairs of c value such that the $s^{c'}[y]$ and $s^{c''}[y]$ functions collide. For some key values, we could even find four byte values c'_1, c'_2, c''_1 and c''_2 such that for each of the 256 possible values of the

c_0 byte, the $s^{c'}[y]$ and $s^{c''}[y]$ functions associated with the $c' = (c_0, c'_1, c'_2)$ and $c'' = (c_0, c''_1, c''_2)$ triplets of bytes collide. This stronger property, which is rather easy to explain using the expression of the b_i constants introduced in Property (2), is not used in the sequel.

The proposed 4 rounds distinguisher uses the collision test derived from property (4) is the following manner :

- select a C set of about 2^{16} c triplet values and a subset of $\{0..255\}$, say for instance a Λ subset of 16 y values.
- for each c triplet value, compute the $L_c = (0E.t_0^c + 0B.t_1^c + 0D.t_2^c + 09.t_3^c)_{y \in \Lambda}$. We claim that such a computation of 16 linear combinations of the outputs represents substantially less than one single Rijndael operation.
- check whether two of the above lists, $L_{c'}$ and $L_{c''}$ are equal. The 4 round distinguisher requires about 2^{20} chosen inputs Y, and since the collision detection computations (based on the analysis of the corresponding T values) require less operations than the 2^{20} 4-inner rounds computations, the complexity of the distinguisher is less than 2^{20} Rijndael encryptions.

Note that property (4) also provides another method to distinguish 4 inner round from a random permutation, using $N \leq 256$ plaintexts and 2^{80} N operations, namely performing an exhaustive search of the 10 unknown constants considered in property (4). Note that a value such as $N = 16$ is far sufficient in practice. However, we only consider in the sequel the above described birthday test, which provides a more efficient distinguisher.

4 An attack of the 7-rounds Rijndael/b=128 cipher with 2^{32} chosen plaintexts

In this Section we show that the 4 inner rounds distinguisher of Section 3 provides attacks of the 7-rounds Rijndael for the b=128 blocksize and the various keysizes. We present two slightly different attacks. The first one (cf Section 4.2 hereafter) is substantially faster than an exhaustive search for the k=196 and k=256 keysizes, but slower than exhaustive search for the k=128 bits keysize. The second attack (cf Section 4.2) is dedicated to the k=128 keysize, and is marginally faster than an exhaustive search for that keysize.

The 7-rounds Rijndael is depicted at Figure 2. X represents a plaintext block, and V represents a ciphertext block. In Figure 2 the 4 inner rounds of Figure 1 are surrounded by one initial $X \rightarrow Y$ round (which consists of an initial key addition followed by one round), and two final rounds (which consist of one $T \rightarrow U$ inner round followed by an $U \rightarrow V$ final round).

Our attack method is basically a combination of the 4-round distinguisher presented in Section 3 and an exhaustive search of some keybytes (or combinations of keybytes) of the initial and the two final rounds. In the attack of Section

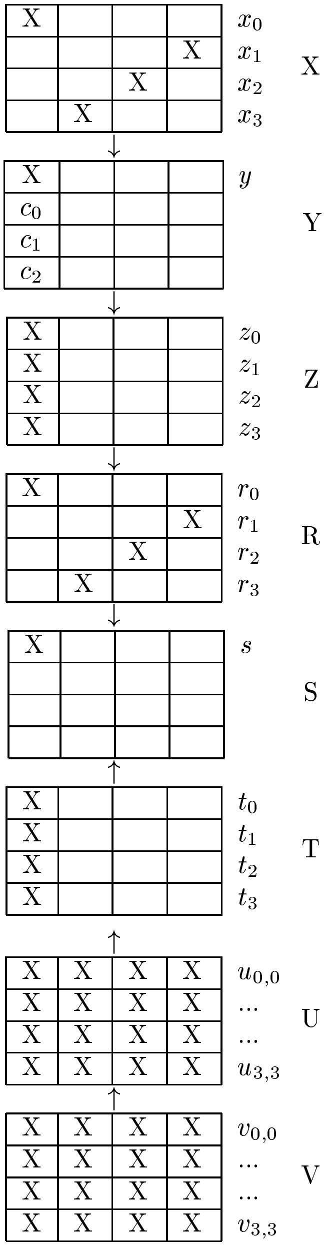


Figure 2: 7-rounds Rijndael

4.1 we are using the property that in the equations provided by the 4-rounds distinguisher there is a variables separations in terms which involve one half of the 2 last rounds key bytes and terms which involve a second half of the 2 last round key bytes in order to save a 2^{80} factor in the exhaustive search complexity. In the attack of Section 4.2, we are using precomputations on colliding pairs of c values to test each 128-bits key assumption with less operations than one single Rijndael encryption.

4.1 An attack of the 7-rounds Rijndael/b=128/k=196 or 256 with 2^{32} chosen plaintexts and a complexity of about 2^{140}

We now explain the attack procedure in some details, using the notation introduced in Figure 2. We fix all X bytes except the four bytes x_0 to x_3 equal to 12 arbitrary constant values. We encrypt the 2^{32} plaintexts obtained by taking all possible values for the x_0 to x_3 bytes, thus obtaining 2^{32} V ciphertext blocks. We are using the two following observations :

Property 5 : If the 4 key bytes added with the x_0 to x_3 bytes in the initial key addition are known (let us denote them by $k_{ini} = (k_0, k_1, k_2, k_3)$, then it is possible to partition the 2^{32} plaintexts in 2^{24} subsets of 256 plaintext values satisfying the conditions of Section 3, i.e. such that the corresponding 256 Y values satisfy the following conditions :

- the y byte takes 256 distinct values (which are known up to an unknown constant first round key byte which is not required for the attack).
- the $c = (c_0, c_1, c_2)$ triplet of bytes is constant ; moreover, each of the 2^{24} subsets corresponds to a distinct c value (the c value corresponding to each subset is known up to three constant first round keybytes which are not required for the attack).
- the 12 other Y bytes are constant and their constant values $Y_{i,j}$ for i=1..3 and j=0..3 is the same for all subsets.

Note that the same property is used in the Rijndael designers' attack.

Property 6 : Each of the t_0, t_1, t_2, t_3 bytes can be expressed as a function of four bytes of the V ciphertext and five unknown key bytes (i.e. 4 of the final round key bytes and one linear combination of the penultimate round key bytes). Therefore, we can "split" the $t^c[y] =' 0E'.t^c_0[y] +' 0B'.t^c_1[y] +' 0D'.t^c_2[y] +' 09'.t^c_3[y]$ combination of the four $t^c_i[y]$ bytes considered in the 4-rounds distinguisher as the XOR of two terms $\tau^c_1[y]$ and $\tau^c_2[y]$ which can both be expressed as a function of 8 ciphertext bytes and 10 unknown key bytes, namely $\tau_1 =' 0E'.t^c_0[y] +' 0B'.t^c_1[y]$ and $\tau_2 =' 0D'.t^c_2[y] +' 09'.t^c_3[y]$. We denote in the sequel by k_{τ_1} those 10 unknown keybytes which allow to derive τ_1 from 8 bytes of the V ciphertext, and by k_{τ_2} those 10 keybytes which allow to derive τ_1 from 8 bytes of the V ciphertext.

We perform an efficient exhaustive search of the k_{ini}, k_{τ_1} and k_{τ_2} keys in the following way :

- For each of the 2^{32} possible k_{ini} assumptions, we can partition the set of the 256^4 possible X values in 256^3 subsets of 256 X values each, according to the value of the c constant, and select say 256^2 of these 256^3 subsets. Thus each of the 256^2 selected subsets is associated with a distinct value of the c constant. Note that the c value associated with a subset and the y values associated with each of the X plaintexts of a subset are only known up to unknown keybits, but this does not matter for our attack. We can denote by $c*$ and $y*$ the known values which only differ from the actual values by fixed unknown key bits.

- Now for each subset associated with a $c*$ constant triplet, based on the say 16 ciphertexts associated with the $y* = 0$ to $y* = 15$ values, we can precompute the $(\tau_1^c(y))_{y*=0..15}$ 16-tuple of bytes for each of the 2^{80} possible k_{τ_1} keys. We can also precompute the $(\tau_2^c(y))_{y*=0..15}$ 16-tuple for each of the 2^{80} possible k_{τ_2} keys.

Based on this precomputation, for each $(c'*, c''*)$ pair of distinct c* values :

- We precompute a (sorted) table the $(\tau_1^{c'}(y) \oplus \tau_1^{c''}(y))_{y*=0..15}$ 16-tuple of bytes for each of the 2^{80} possible k_{τ_1} keys (the computation of each 16-tuple just consists in xoring two precomputed values)

- For each of the 2^{80} possible values of the k_{τ_2} key, we compute the $(\tau_2^{c'}(y) \oplus \tau_2^{c''}(y))_{y*=0..15}$ 16-tuple of bytes associated with the observed ciphertext, and check whether this t-uple belongs to the precomputed table of 16-tuple $(\tau_1^{c'}(y) \oplus \tau_1^{c''}(y))_{y*=0..15}$. If for a given k_{τ_1} value there exists a k_{τ_2} value such that $(\tau_1^{c'}(y) \oplus \tau_1^{c''}(y))_{y*=0..15} = (\tau_2^{c'}(y) \oplus \tau_2^{c''}(y))_{y*=0..15}$, (i.e. $t^{c'}[y] = t^{c''}[y]$ for each of the y values associated with $y* = 0$ to 16, this represents an alarm). The equality between the t bytes associated with c' and c'' is checked for the other $y*$ values. If the equality is confirmed, this means that a collision between the $s^c[y]$ functions associated with c' and c''. This provides 20 bytes of information on the last and penultimate key values, since with overwhelming probability, the right values of k_{ini}, k_{τ_1} and k_{τ_2} have then been found.

Since the above procedure tests whether the exist collisions inside a random set of 256^2 of the 256^4 possible $s^c[y]$ functions, the probability of the procedure to result in a collision, and thus to provide k_{ini}, k_{τ_1} and k_{τ_2} is high (say about 1/2). In other words, the success probability of the attack is about 1/2.

Once k_{ini}, k_{τ_1} and k_{τ_2} have been found, the 16-bytes final round key is entirely determined and the final round can be decrypted, so one is left with the problem of cryptanalysing the 6-round version of Rijndael. One might object that the last round of the left 6-round cipher is not a final round, but an inner round. However, it is easy to see that by applying a linear one to one change of variable to U and the 6th round key (i.e. replacing U by a U' linear function of U and K_6 by a linear function K'_6 of K_6), the last round can be represented as a final round (i.e. U' is the image of T by the final round associated with

K'_6). Thus we are in fact left with the cryptanalysis of the 6-round Rijndael, and the last round subkey is easy to derive. The process of deriving the subkeys of the various rounds can then be continued (using a subset of the 2^{32} chosen plaintexts used for the derivation of k_{ini}, k_{τ_1} and k_{τ_2}), with negligible additional complexity, until the entire key has eventually been recovered.

4.2 An attack of the 7-rounds Rijndael/b=128/k=128 2^{32} chosen plaintext

We now outline a variant of the former attack that is dedicated to the k=128 bits version of Rijndael and is marginally faster than an exhaustive search. This attack requires a large amount of precomputations.

As a matter of fact, it can be shown that the 4 c-dependent bytes that determine the mapping between four $z_i^c[y]$ bytes and the four $r_i^c[y]$ are entirely determined by 12 key-dependent (and c-independent) bytes. For each of the 256^{12} possible values of this $\phi(K)$ 12-tuple of bytes, we can compute colliding c' and c'' triplets of bytes (this can be done performing about 256^2 partial Rijndael computations corresponding to say 256^2 distinct c values and looking for a collision. One can accept that no collision be found for some $\phi(K)$ values : this just means that the attack will fail for a certain fraction (say 1/2) of the key values. We store c' and c'' (if any) in a table of 256^{12} $\phi(K)$ entries.

Now we perform an exhaustive search of the K key. To test a key assumption, we compute the k_{ini}, k_{τ_1}, k_{τ_2} and $\phi(K)$ values. Then we find the (c', c'') pair of colliding c values in the precomputed table, compute the two associated $c'*$ and c'' values, determine which two precomputed lists $(V^{c'}[y])_{y*=0..15}$ $(V^{c''}[y])_{y*=0..15}$of 16 ciphertext values each are associated with $c'*$ and $c''*$, and finally compute the associated $(t^{c'}[y])_{y*=0..15}$ and $(t^{c''}[y])_{y*=0..15}$ bytes by means of partial Rijndael decryption. The two values associated with $y* = 0$ are first computed and compared. The two values associated with $y* = 1$ are only computed and compared if they are equal, etc, thus in average only two partial decryption are performed. It the two lists of 16 t bytes are equal, then there is an alarm, and the K is further tested with a few plaintexts and ciphertexts.

We claim than the complexity of the operations performed to test one K key is marginally less than one Rijndael encryption.

5 Conclusion

We have shown that the existence of collisions between some partial byte oriented functions induced by the Rijndael cipher provides a distinguisher between 4 inner rounds of Rijndael and a random permutation, which in turn enables to mount attacks on a 7-rounds version of Rijndael for any key-length.
Unlike many recent attacks on block ciphers, our attacks are not statistical in

nature. They exploit (using the birthday paradox) a new kind of cryptanalytic bottleneck, namely the fact that a partial function induced by the cipher (the $s^c[y]$ function) is entirely determined by a remarkably small number of unknown constants. Therefore, unlike most statistical attacks, they require a rather limited number of plaintexts (about 2^{32}). However, they are not practical because of their high computational complexity, and do not endanger the full version of Rijndael. Thus we do not consider they represent arguments against Rijndael in the AES competition.

References

[AES99] http://www.nist.gov/aes

[DaKnRi97] J. Daemen, L.R. Knudsen and V. Rijmen, "The Block Cipher Square". In *Fast Software Encryption - FSE'97* , pp. 149-165, Springer Verlag, Haifa, Israel, January 1997.

[DaRi98] J. Daemen, V. Rijmen, "AES Proposal : Rijndael", *The First Advanced Encryption Standard Candidate Conference,* N.I.S.T., 1998.

Relationships among Differential, Truncated Differential, Impossible Differential Cryptanalyses against Word-Oriented Block Ciphers like Rijndael, E2

Makoto Sugita[1], Kazukuni Kobara[2], Kazuhiro Uehara[1], Shuji Kubota[1], Hideki Imai[2]
[1] NTT Wireless Systems Innovation Laboratory, Network Innovation Laboratories,
1-1 Hikari-no-oka, Yokosuka-shi, Kanagawa, 239-0847 Japan
E-mail: {sugita, uehara, kubota}@wslab.ntt.co.jp
[2] Institute of Industrial Sciences, The University of Tokyo
Roppongi, Minato-ku, Tokyo 106-8558, Japan
E-mail:{kobara, imai}@imailab.iis.u-tokyo.ac.jp

Abstract

We propose a new method for evaluating the security of block ciphers against differential cryptanalysis and propose new structures for block ciphers. To this end, we define the word-wise Markov (Feistel) cipher and random output-differential (Feistel) cipher and clarify the relations among the differential, the truncated differential and the impossible differential cryptanalyses of the random output-differential (Feistel) cipher. This random output-differential (Feistel) cipher model uses a not too strong assumption because denying this approximation model is equivalent to denying truncated differential cryptanalysis. Utilizing these relations, we evaluate the truncated differential probability and the maximum average of differential probability of the word-wise Markov (Feistel) ciphers like Rijndael, E2 and the modified version of block cipher E2. This evaluation indicates that all three are provably secure against differential cryptanalysis, and that Rijndael and a modified version of block cipher E2 have stronger security than E2.

keywords. truncated differential cryptanalysis, truncated differential probability, maximum average of differential probability, generalized E2-like transformation, SPN-structure, word-wise Markov cipher, random output-differential cipher

1 Introduction

As a measure of the security of block ciphers, the maximum average of differential probability was defined by Nyberg and Knudsen [15] by generalizing provable security against differential cryptanalysis as introduced by Biham and Shamir [2]. Based on this idea, many new block ciphers have been proposed, e.g. the block cipher MISTY was proposed by M. Matsui [10]. It was designed on the basis of the theory of provable security against differential and linear cryptanalysis.

The block cipher E2 was proposed in [6] as an AES candidate. This cipher uses Feistel structures as a global structure like DES, and uses the SPN (Substitution and Permutation Network)-structure in its S-boxes. [6] said this cipher can be 'proved' to offer immunity against differential cryptanalysis by counting the maximum number of active S-boxes. However, Sugita proposed a method for evaluating the maximux average of differential probability of SPN-structures, and then evaluated the SPN-structure of E2[16, 17]. Using the similar method, Matsui stated that 8-rounds E2 can be defeated by truncated differential cryptanalysis [19, 14], which implies that just counting the maximum number of active S-boxes is not sufficient for proving the security of block ciphers.

The block cipher Rijndael was also proposed as an AES candidate [3]. This cipher uses the SPN (Substitution and Permutation Network)-structure as its basic structure. The basis for proving its security against differential cryptanalysis involves a similar evaluation method as used for E2. Therefore more accurate proof is necessary.

In this paper, we introduce the word-wise Markov (Feistel) cipher and random output-differential (Feistel) cipher as a approximation model for the accurate definition of truncated differential probability, and clarify the relationships among differential, truncated differential and impossible differential cryptanalyses, and propose a new method for evaluating the security of block ciphers against differential, truncated differential and impossible differential cryptanalysis under this model, and propose new structures for block ciphers that are secure against these cryptanalyses. This random output-differential (Feistel) cipher model does not use too strong an assumption because denying this model is equivalent to denying truncated differential cryptanalysis.

This report is organized as follows.

Section 2 defines the structures of word-oriented block ciphers like SPN-Structures, PSN-structures and the E2($'$)-like transformation.

Section 3 defines the differential probability, and defines the word-wise Markov (Feistel) cipher, random output-differential (Feistel) cipher, and using these definitions, defines the truncated differential probability.

Section 4 clarifies the relations between the truncated differential probability and the differential probability of the random output-differential (Feistel) cipher. It then describes a procedure for calculating the truncated differential probability and (maximum average of) the differential probability of typical random output-differential ciphers like SPN-structures including Rijndael and E2($'$)-like transformations. It proves that both Rijndael and the modified E2-like transformation are provably secure against differential, truncated differential and impossible differential cryptanalysis if they can be approximated as random output-differential (Feistel) ciphers.

Section 5 concludes this paper.

2 Structures of Word-oriented Block Ciphers

2.1 Word-oriented Block Ciphers

A word-oriented block cipher is a block cipher whose input and output data is a set of input words of fixed size, and whose operations consist only of word-wise operations of fixed size. In the usual case, the word size is 8, i.e. byte size. Example of these ciphers include Rijndael, E2, etc.

2.2 Feistel Structures

Associate with a function $f : GF(2)^n \to GF(2)^n$, a function $\delta_{2n,f}(L, R) = (R \oplus f(L), L)$ for all $L, R \in GF(2)^n$. $\delta_{2n,f}$ is called the Feistel transformation associated with f. Furthermore, for functions $f_1, f_2, \cdots, f_s : GF(2)^n \to GF(2)^n$, define $\psi_n(f_1, f_2, \cdots, f_s) = \delta_{2n,f_s} \circ \cdots \circ \delta_{2n,f_2} \circ \delta_{2n,f_1}$. We call $D(f_1, f_2, \cdots, f_s) = \psi_n(f_1, f_2, \cdots, f_s)$ as the s-round Feistel structure. At this time, we call the functions $f_1, f_2, \cdots, f_s$ as S-boxes of the Feistel structure $D(f_1, f_2, \cdots, f_s)$.

2.3 SPN-Structures and PSN-Structures

[11] defines SPN-Structures. First we define the 3-layer SPN-structure.

This structure consists of two kinds of layers, i.e. nonlinear layer and bijective linear layer. Each layer has different features as follows.

Nonlinear layer: This layer is composed of m parallel n-bit bijective nonlinear transformations.

Linear layer: This layer is composed of linear transformations over the field $GF(2^n)$ (especially in the case of E2, bit-wise XOR), where inputs are transformed linearly to outputs per word (n-bits).

Next for $s \in \mathbf{N}$ we define the s-layer SPN-structure, which consists of s layers. First is a nonlinear layer, second is a linear layer, third is a nonlinear layer, $\cdots$.

Similarly, for $s \in \mathbf{N}$ we define the s-layer PSN-structure. This layer consists of s layers. First is a linear layer, second is a nonlinear layer, third is a linear layer, $\cdots$.

The SPN-structure is the basic structure of Rijndael, a candidate for AES. We will analyze the security of Rijndael afterwards.

2.4 E2($'$)-like Transformations

[6] proposed the block cipher E2. This cipher has Feistel structures and its S-boxes are composed of 3-layers SPN structures. We generalize this structure, and define E2-like transformations as Feistel structure with S-boxes composed of s-layers (in the case of E2, 3-layers) SPN-structures.

Similarly, we define E2$'$-like transformations as Feistel structures with S-boxes composed of s-layer PSN-structures.

3 Differential Probability, Truncated Differential Probability, Word-wise Markov (Feistel) Cipher and Random Output-Differential (Feistel) Cipher

This section defines the (maximum average of) differential probability, truncated differential probability, word-wise (Feistel) Markov cipher and random output-differential (Feistel) cipher.

3.1 Differential Probability of Block Ciphers

We define the differential of block ciphers. We consider the encryption of a pair of distinct plaintexts by an r-round iterated cipher. Here the round function $Y = f(X, Z)$ is such that, for every round sub-key Z, $f(\ \cdot\ , Z)$ establishes a one-to-one correspondence between the round input X and the round output Y. Let the "difference" ΔX between two plain-texts (or two cipher texts) X and X^* be defined as

$$\Delta X = X \oplus X^*.$$

From the pair of encryption results, one obtains the sequence of differences $\Delta X(0), \Delta X(1), \cdots, \Delta X(r)$ where $X(0) = X$ and $X(0)^* = X^*$ denote the plaintext pair (such that $\Delta X(0) = \Delta X$) and where $X(i)$ and $X^*(i)$ for $(0 < i < r)$ are the outputs of the i-th round, which are also the inputs to the $(i+1)$-th round. The sub-key for the i-th round is denoted as $Z^{(i)}$.

Next we define the i-th round differential and maximum average of differential probabilities.

Definition 1 *[7] An i-round differential is the couple (α, β), where α is the differential of a pair of distinct plaintexts X and X^* and β is a possible difference for the resulting i-th round outputs $X(i)$ and $X^*(i)$. The probability of an i-round differential (α, β) is the conditional probability that β is the difference, $\Delta X(i)$, of the cipher text pair after i rounds given that the plaintext pair (X, X^*) has difference $\Delta X = \alpha$ when the plaintext X and the sub-keys $Z^{(1)}, \cdots, Z^{(i)}$ are independent and uniformly random. We denote this differential probability by $P(\Delta X(i) = \beta | \Delta X = \alpha)$.*

The probability of an s-round differential is known to satisfy the following property.

Lemma 1 *[7] For the Markov cipher, the probability of an s-round differential equals*

$$P(\Delta X(s) = \beta(s) | \Delta X(0) = \beta(0)) =$$
$$\sum_{\beta(1)} \sum_{\beta(2)} \cdots \sum_{\beta(s-1)} \prod_{i=1}^{s} P(\Delta X(i) = \beta(i) | \Delta X(i-1) = \beta(i-1)).$$

We define the maximum average of differential probability as follows. This value is known to be the best measure with which to confirm that block ciphers are secure against differential cryptanalysis.

Definition 2 *[15] We define the maximum average of differential probability $ADP^{(s)}_{\max}$ by*

$$\mathrm{ADP}^{(s)}_{\max} = \max_{\alpha \neq 0, \beta} P(\Delta X(s) = \beta | \Delta X = \alpha).$$

3.2 Word-wise Markov (Feistel) Cipher

[5] uses the truncated differential for the cryptanalysis of word-oriented block ciphers. However, the accurate definition of truncated differential probability is not offered because this cryptanalysis is essentially based on approximation. In this subsection, in order to legitimate this notion, we redefine the truncated differential probability of word-oriented block ciphers.

We consider the encryption of a pair of distinct plaintexts by an r-round iterated cipher. Here the round function $Y = f(X, Z)$ is such that, for every round sub-key $Z = (Z_1, Z_2, \cdots, Z_{m'}) \in GF(2^n)^{m'}$, $f(\ \cdot\ , Z)$ establishes a one-to-one correspondence between the round input $X = (X_1, X_2, \cdots, X_m) \in GF(2^n)^m$ and the round output $Y = (Y_1, Y_2, \cdots, Y_m) \in GF(2^n)^m$.

We define a characteristic function $\chi : GF(2^n)^m \to GF(2)^m$, $(x_1, \cdots, x_m) \longmapsto (y_1, \cdots, y_m)$ by

$$y_i = \begin{cases} 0 & \text{if } x_i = 0 \\ 1 & \text{otherwise,} \end{cases}$$

Hereafter, we call $\chi(x)$ as a characteristic of $x \in GF(2^n)^m$.

For the definition of the truncated differential probability, we define the word-wise Markov cipher as a real block-cipher model, in the same way as the Markov cipher was in [7]

Definition 3 *A word-oriented cipher with round function $Y = f(X, Z)(X = (X_1, X_2, \cdots, X_m) \in GF(2^n)^m$, $Y = (Y_1, Y_2, \cdots, Y_m) \in GF(2^n)^m, Z = (Z_1, Z_2, \cdots, Z_{m'}) \in GF(2^n)^{m'})$, is a word-wise Markov cipher if for all choices of $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_m) \in GF(2^n)^m (\alpha \neq 0)$, $\beta = (\beta_1, \beta_2, \cdots, \beta_m) \in GF(2^n)^m (\beta \neq 0)$ and $p \in \{1, 2, \cdots, m'\}$,*

$$P(\Delta Y_p = \beta_p | \Delta X = \alpha, X = \gamma)$$

is independent of γ, and $P(\Delta Y_p = \beta_p | \Delta Y_p \neq 0, \Delta X = \alpha, X = \gamma)$ $(p = 1, 2, \cdots, m)$ are jointly statistically independent when the sub-key Z is uniformly random, or, equivalently, if

$$P(\Delta Y_p = \beta_p | \Delta X = \alpha, X = \gamma) = P(\Delta Y_p = \beta_p | \Delta X = \alpha)$$

for all choices of γ and $P(\Delta Y_p = \beta_p | \Delta Y_p \neq 0, \Delta X = \alpha)$ $(p = 1, 2, \cdots, m)$ are jointly statistically independent when the sub-key Z is uniformly random, where $\Delta X = (\Delta X_1, \Delta X_2, \cdots, \Delta X_m)$, $\Delta Y = (\Delta Y_1, \Delta Y_2, \cdots, \Delta Y_m)$ are the differential of X, Y, respectively.

Example. the PSN-structure is a word-wise Markov cipher, if every bijective nonlinear function in a nonlinear layer consists of a concatenation of XOR and substitution (like DES does). Therefore, block cipher Rijndael and the S-boxes of block cipher E2 are also word-wise Markov ciphers with the same kind of nonlinear functions.

We expand this definition to the Feistel cipher.

Definition 4 *We define a word-wise Markov Feistel cipher as a Feistel cipher whose S-boxes are word-wise Markov ciphers.*

Example. E2′-like transformation is a word-wise Markov Feistel cipher because the PSN-structure is a word-wise Markov cipher if every nonlinear function in a nonlinear layer consists of the concatenation of XOR of the key and substitution (like DES does).

3.3 Random Output-Differential (Feistel) Cipher

As preparation for defining the random output-differential cipher, we define the random output-differential transformation.

Definition 5 *A word-oriented transformation* $Y = g(X, Z)$ $(X = (X_1, X_2, \cdots, X_m) \in GF(2^n)^m$, $Y = (Y_1, Y_2, \cdots, Y_m) \in GF(2^n)^m$, $Z = (Z_1, Z_2, \cdots, Z_{m'}) \in GF(2^n)^{m'})$, *is a random output-differential transformation, if for any input-differential value* α, *the following relation is satisfied,*

$$P(\Delta Y = \beta | \Delta X = \alpha) = p^{\mathrm{h}(\chi(\beta))} P(\chi(\Delta Y) = \chi(\beta) | \Delta X = \alpha),$$

when keys are randomly selected, where h *is the function that indicates the Hamming weight of the input value,* $p = 1/(2^n - 1)$, *and* $\Delta X = (\Delta X1, \Delta X_2, \cdots, \Delta X_m)$, $\Delta Y = (\Delta Y_1, \Delta Y_2, \cdots, \Delta Y_m)$ *are the differential of* X, Y, *respectively.*

Using this definition, we define the random output-differential cipher for word-oriented block cipher as approximation model of word-wise Markov cipher.

Definition 6 *A word oriented cipher with round functions* $X(i + 1) = f(X(i), Z^{(i)})(i = 0, 1, \cdots, r - 1)$, *where* $Z^{(i)}(i = 0, 1, \cdots)$ *are sub-keys, is a random output-differential cipher if for any random output-differential transformation* $X(0) = g(X, Z^{(0)})$, *the composite transformation* $X(1) = f(g(X, Z^{(0)}), Z^{(1)})$ *is also a random output-differential transformation.*

At this time, we call a round function which composes a random output-differential cipher by concatenating, as random output-differential round function.

We expand this definition to the Feistel cipher.

Definition 7 *A Feistel cipher with S-boxes* $Y = f(X, Z^{(i)})$ $(i = 0, 1, \cdots)$, *where* $Z^{(i)}(i = 0, 1, \cdots)$ *are sub-keys and i-th round output is* $X(i) = (X(i)_L, X(i)_R)$, *is a random output-differential Feistel cipher, if its S-boxes are random output-differential ciphers and the round function of the Feistel cipher*

$$(X(i + 1)_L, X(i + 1)_R) = (X(i)_R, X(i)_L \oplus f(X(i)_R, Z^{(i)}))$$

is a random output-differential round function.

Matsui stated in his presentation of [14] that 8-round E2 can be cryptanalyzed by truncated differential cryptanalysis only assuming randomness of keys. However, this is not accurate, because he tacitly assumes this random output-differential cipher as an approximation model of E2 in his explanation.

However, this approximation may be effective for word-wise Markov (Feistel) cipher like E2, E2′-like transformation and Rijndael. In fact, in the case of E2′-like transformation with 2-layer PSN-structures, which is also a word-wise Markov Feistel cipher for example, let the $\Delta X \in GF(2^n)^{2m}$ be a input differential of this cipher, if the input-differential of S-box $\Delta W = (\Delta W_1, \Delta W_2, \cdots, \Delta W_m) \in GF(2^n)^m(\chi(\Delta W) = \gamma' \in GF(2)^m)$ is randomly distributed with the probability $P(\Delta W = \gamma | \chi(\Delta W) = \gamma', \Delta X = \alpha) = p^{\mathrm{h}(\gamma')}$ for all $\gamma = (\gamma_1, \gamma_2, \cdots, \gamma_m)$ (where $\chi(\gamma) = \gamma'$), then the output-differential of S-box $\Delta U = (\Delta U_1, \Delta U_2, \cdots, \Delta U_m) \in GF(2^n)^m$ $(\chi(\Delta U) = \delta' \in GF(2)^m)$ is supposed to be approximately random, i.e. approximately $P(\Delta U = \delta | \chi(\Delta U) = \delta', \Delta X = \alpha) = p^{\mathrm{h}(\delta')}$, where $\delta = (\delta_1, \delta_2, \cdots, \delta_m), \chi(\delta) = \delta'$ because, for the input-differential of nonlinear layer $\Delta W = (\Delta W_1, \Delta W_2, \cdots, \Delta W_m) \in GF(2^n)^m$, each $P(\Delta W_p = \gamma_p | \chi(\Delta W) = \gamma', \Delta X = \alpha) = p = 1/(2^n - 1)$ implies $P(\Delta U_p = \delta_p | \chi(\Delta U) = \delta', \Delta X = \alpha) = p = 1/(2^n - 1)$ and each $P(\Delta U_p = \delta_p | \Delta W_p = \gamma_p \neq 0, \Delta X = \alpha)$ $(\gamma = (\gamma_1, \gamma_2, \cdots, \gamma_m), \chi(\gamma) = \gamma')$ are jointly statistically independent from the definition of word-wise Markov cipher.

So we use this random output-differential cipher as an effective approximation model in the following discussion.

Note. Matsui assumed the randomness of input-differential of nonlinear layers $\Delta W = (\Delta W_1, \cdots, \Delta W_m)$, i.e.

$$P(\Delta W = \gamma | \chi(\Delta W) = \gamma', \Delta X = \alpha) = p^{\mathrm{h}(\gamma')} P(\chi(\Delta W) = \gamma' | \Delta X = \alpha)$$

instead of the randomness of output-differential of nonlinear layers $\Delta U = (\Delta U_1, \cdots, \Delta U_m)$, i.e.

$$P(\Delta U = \beta | \chi(\Delta U) = \beta', \Delta X = \alpha) = p^{\mathrm{h}(\beta')} P(\chi(\Delta U) = \beta' | \Delta X = \alpha)$$

in his presentation of [14]. The randomness of ΔW is a stronger assumption than the randomness of ΔU, because, in the case of E2$'$-like transformation with 2-layer PSN-structures for example, the randomness of ΔW also implies the randomness of ΔU. Furthermore, the randomness of ΔW may be too strong or even nonsense, because the randomness of input-differentials of linear layer $\Delta W = (\Delta W_1, \cdots, \Delta W_m)$ do not always yield the randomness of input-differentials of nonlinear layer $\Delta U = (\Delta U_1, \cdots, \Delta U_m)$: Two input-differential words of nonlinear layers ΔW_{p_1}, ΔW_{p_2} $(p_1 \neq p_2)$ may be both random, i.e.

$$P(\Delta W_{p_1} = \gamma_{p_1} | \Delta X = \alpha) = P(\Delta W_{p_2} = \gamma_{p_2} | \Delta X = \alpha) = p = 1/(2^n - 1)$$

but coincide, i.e. constantly $\Delta W_{p_1} = \Delta W_{p_2}$.

Therefore, we interpret Matsui's tacit assumption in his explanation as a random output-differential (Feistel) cipher.

3.4 Truncated Differential Probability

Using these definitions, we can accurately define the truncated differential probability. In this definition, as a cipher model, we consider a cipher with a random output-differential initial transformation $X(0) = g(X, Z^{(0)})$, and a random output-differential round function $X(i+1) = f(X(i), Z^{(i)})(i = 0, 1, \cdots, r-1)$ where $Z^{(i)}(i = 0, 1, \cdots)$ are sub-keys.

Definition 8 *Let $X(0) = g(X, Z^{(0)})$ be an arbitrary random output-differential initial transformation and $X(i+1) = f(X(i), Z^{(i)})$ be a round function such that $X(r) = (f \circ \cdots \circ f \circ g)(X, Z^{(0)}, Z^{(1)}, \cdots, Z^{(r)})$ is also a random output-differential cipher for all r. An i-round truncated differential of i-round iterated cipher $X(r) = (f \circ \cdots \circ f)(X(0), Z^{(1)}, \cdots, Z^{(r)})$ is the couple (α', β'), where α is the differential of a pair of distinct values $X(0)$ and $X^*(0)$, $\alpha' = \chi(\alpha)$ is the characteristic of α; β is a possible difference for the resulting i-th round outputs $X(i)$ and $X^*(i)$; $\beta' = \chi(\beta)$ is the characteristic of β. The probability of i-round truncated differential (α', β') is the conditional probability that β' is the characteristic of difference $\Delta X(i)$ of the cipher text pair after i rounds given that the characteristic of pair $(X(0), X(0)^*)$ has difference $\chi(\Delta X) = \alpha'$ when the plaintext X and the sub-keys $Z^{(0)}, \cdots, Z^{(i)}$ are independent and uniformly random. We denote this truncated differential probability by $P'_i(\beta'(i), \beta'(0)) = P(\chi(\Delta X(i)) = \beta'(i) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha)$.*

This definition is well defined if we assume the random output-differential (Feistel) cipher. Without the assumption, this is not well-defined, because two input-differential values with same characteristic value do not always yield the same truncated differential probabilities. We assume this model as an effective approximation model of a word-wise Markov cipher.

4 Truncated Differential Probability and Differential Probability of Random Output-Differential (Feistel) Ciphers

4.1 Truncated Differential Probability of PSN-structures and Differential Probability of SPN-structures

In this subsection, we evaluate the truncated differential probability of the $2s$ layer PSN-structure and (the maximum average of) the differential probability of the $(2s+1)$ layer SPN-structure, where we assume all random functions are bijective. In this calculation, we first calculate the truncated differential probability of the $2s$ layer PSN-structure, and, using this probability, we calculate (the maximum average of) the differential probability of the $(2s+1)$ layer SPN-structure.

We assume the first nonlinear layer is a random output-differential (initial) transformation, and the round functions, which are composed of a linear layer and a nonlinear layer, i.e. 2-layer PSN-structures, is a random output-differential round function. We denote ΔX as the input-differential of the first nonlinear layer, $\Delta X(0)$ as the output-differential of the first nonlinear layer, $\Delta X(1)$ as the output-differential of the second nonlinear layer, $\cdots$, $\Delta X(s)$ as the output-differential of $(s+1)$-th nonlinear layer, $\Delta Y(0)$ as the input-differential of the first nonlinear layer, $\Delta Y(1)$ as the input-differential of the second nonlinear layer, $\cdots$, $\Delta Y(s)$ as the input-differential of the $(s+1)$-th nonlinear layer.

We denote the differential probability of $(2s+1)$-layer SPN-structures as

$$P_i(\beta(i), \alpha) = P(\Delta X(i) = \beta(i) | \Delta X = \alpha).$$

We denote the truncated differential probability of $2s$-layer PSN-structures as

$$P'_i(\beta'(i), \beta'(0)) = P(\chi(\Delta X(i)) = \beta'(i) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha).$$

The relation between differential probability and truncated differential probability can be represented as follows, where $\beta'(i) = \chi(\beta(i))$ for all $i = 0, 1, \cdots, s$,

$$\begin{aligned} &P_i(\beta(i), \alpha) = \\ &\quad \sum_{\beta'(0)} P(\Delta X(i) = \beta(i)) | \chi(\Delta X(i)) = \beta'(i), \Delta X = \alpha) \\ &\quad * \ P'_i(\beta'(i), \beta'(0)) * P(\chi(\Delta X(0) = \beta'(0)) | \Delta X = \alpha). \end{aligned}$$

In this case, if we assume the initial transformation is a random output-differential transformation and

$$P(\chi(\Delta X(0)) = \beta'(0) | \Delta X = \alpha) = \begin{cases} 1 & \text{if } \beta'(0) = \chi(\alpha) \\ 0 & \text{otherwise,} \end{cases}$$

as a natural approximation model of the first nonlinear layer, we can prove

$$P_i(\beta(i), \alpha) = p^{\mathrm{h}(\beta'(i))} P'_i(\beta'(i), \alpha'),$$

because

$$P(\Delta X(i) = \beta(i) | \chi(\Delta X(i)) = \beta'(i), \Delta X = \alpha) = p^{\mathrm{h}(\beta'(i))},$$

from the assumption of random output-differential cipher, where $p = 1/(2^n - 1)$ and h is the function that indicates the Hamming weight of the input value.

This relation clearly indicates the relationship between the differential probability and the truncated differential probability. From this relation we can easily calculate the differential probability from the truncated differential probability in the case of random output-differential cipher. This relation also implies that the possibility of truncated differential cryptanalysis is equivalent to the possibility of differential cryptanalysis, because the ratio of obtained probability to average probability do not change.

4.2 Procedure for Calculating Differential and Truncated Differential Probability of the SPN-structure

The procedure for calculating truncated differential probability and the maximum average of the differential probability in case of the SPN structure is as follows.

For this procedure, we define function $\mathrm{N}(P, \gamma, \delta)$ for $m \times m$ matrix P over $GF(2^n)$ and $\gamma, \delta \in GF(2)^m$ by

$$\begin{gathered} \mathrm{N}(P, \gamma, \delta) = \#\{(\Delta X, \Delta Y) \in (GF(2^n)^m)^2 \setminus \{0\} | \\ \Delta Y = P\Delta X, \chi(\Delta X) = \gamma, \chi(\Delta Y) = \delta\}, \end{gathered}$$

For this calculation we define semi-order $\prec$ in $GF(2)^m$ as follows.

$$a \prec b \Leftrightarrow (\forall i; (a(i) = 1 \Rightarrow b(i) = 1)) \wedge (a \neq b)$$

where we denote $a(i)$ and $b(i)$ as the i-th significant bits of a and b, respectively.

For $m \times m$ matrix P over $GF(2^n)$ and $\gamma, \delta \in GF(2)^m$, we define

$$\begin{aligned} \mathrm{M}(P, \gamma, \delta) = \#\{(\Delta X, \Delta Y) \in (GF(2^n)^m)^2 \setminus \{0\} | \\ \Delta Y = P\Delta X, \chi(\Delta X) \preceq \gamma, \chi(\Delta Y) \preceq \delta\}, \end{aligned}$$

and $\mathrm{N}(P, \gamma, \delta)$ can be calculated recursively, using the following relations.

$$\mathrm{N}(P, \gamma, \delta) = \mathrm{M}(P, \gamma, \delta) - \sum_{(\gamma', \delta') \prec (\gamma, \delta)} \mathrm{N}(P, \gamma', \delta')$$

In this case, we assume a random output-differential cipher. Under this assumption, we can prove the following lemma.

Lemma 2

$$\begin{aligned} & P_i'(\beta'(i), \beta'(0)) = \\ & \sum_{\beta'(i-1)} \mathrm{N}(P, \beta'(i), \beta'(i-1)) p^{\mathrm{h}(\beta'(i-1))} P_{i-1}'(\beta'(i-1), \beta'(0)), \end{aligned}$$

where $p = 1/(2^n - 1)$*.*

Proof. From the assumption of a random output-differential cipher,

$$\begin{aligned} & P(\Delta X(i-1) = \beta(i-1) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha) \\ = \ & P(\Delta X(i-1) = \beta(i-1) | \chi(\Delta X(i-1)) = \beta'(i-1), \Delta X = \alpha) \\ * \ & P(\chi(\Delta X(i-1)) = \beta'(i-1) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha) \\ = \ & p^{\mathrm{h}(\beta'(i))} P(\chi(\Delta X(i-1)) = \beta'(i-1) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha) \\ = \ & p^{\mathrm{h}(\beta'(i-1))} P_{i-1}'(\beta'(i-1), \beta'(0)), \end{aligned}$$

where $\beta'(i-1) = \chi(\beta(i-1))$.

From the definition of N,

$$\begin{aligned} \mathrm{N}(P, \beta'(i), \beta'(i-1)) = \#\{(\Delta X(i), \Delta X(i-1)) \in (GF(2^n)^m \setminus \{0\})^2 | \\ \Delta X(i) = P\Delta X(i-1), \chi(\Delta X(i)) = \beta'(i), \chi(\Delta X(i-1)) = \beta'(i-1)\}, \end{aligned}$$

.

Therefore,

$$\begin{aligned} & P_i'(\beta'(i), \beta'(0)) \\ & = \sum_{\beta'(i-1)} \mathrm{N}(P, \beta'(i), \beta'(i-1)) P(\Delta X(i-1) = \beta(i-1) | \chi(\Delta X(0)) = \beta'(0)) \\ & = \sum_{\beta'(i-1)} \mathrm{N}(P, \beta'(i), \beta'(i-1)) p^{\mathrm{h}(\beta'(i-1))} P_{i-1}'(\beta'(i-1), \beta'(0)), \end{aligned}$$

This lemma, yields the following procedure.

1) **Computing the Number of All Differential Paths**
For given P, calculate $N(P, \gamma, \delta)$ for every $\gamma, \delta \in GF(2)^m$.
$\mathrm{M}(P, \gamma, \delta)$ can be easily calculated by simple rank calculation as follows.

$$\begin{aligned}
&\mathrm{M}(P,\gamma,\delta)\\
&= \#\{(\Delta X,\Delta Y)\in GF(2^n)^{2m}\setminus\{0\}|P\Delta X=\Delta Y, F(\bar\gamma)\Delta X=0, F(\bar\delta)\Delta Y=0\}\\
&= 2^{n*\dim\{(\Delta X,\Delta Y)\in GF(2)^{2m}\setminus\{0\}|P\Delta X=\Delta Y,F(\bar\gamma)\Delta X=0,F(\bar\delta)\Delta Y=0\}}-1\\
&= 2^{n(2m-\mathrm{rank}(\begin{pmatrix} P & E\\ F(\bar\gamma) & O\\ O & F(\bar\delta)\end{pmatrix}))}-1,
\end{aligned}$$

where $\bar\gamma$ and $\bar\delta$ are the complements of γ and δ, respectively, E is an identity matrix, and $F(\bar\gamma)$, $F(\bar\delta)$, denote the diagonal matrices over $GF(2^n)$ whose (i,i) component equals the i-th significant bit of $\bar\gamma$, $\bar\delta$ for $i=1,\cdots,m$, respectiveley.
$\mathrm{N}(P,\gamma,\delta)$ can be calculated recursively from the values of $\mathrm{M}(P,\gamma,\delta)$, using the following relation.

$$\mathrm{N}(P,\gamma,\delta)=\mathrm{M}(P,\gamma,\delta)-\sum_{(\gamma',\delta')\prec(\gamma,\delta)}\mathrm{N}(P,\gamma',\delta')$$

2) **Initialization**
For given $\alpha'\in GF(2)^m$, calculate $P_0'(\beta'(0),\alpha')$ for every $\beta'(0)\in GF(2)^m$, where

$$P_0'(\beta'(0),\alpha')=\begin{cases}1 & \text{if } \beta'(0)=\alpha'\\ 0 & \text{otherwise,}\end{cases}$$

3) **Recursive Computation of Truncated Differential Probability**
Utilizing the values of $N(P,\gamma,\delta)$, calculate $P_i'(\beta'(i),\alpha')$ recursively for every $\beta'(i)\in GF(2)^m$.

$$\begin{aligned}&P_i'(\beta'(i),\alpha')=\\ &\sum_{\beta'(i-1)}\mathrm{N}(P,\beta'(i),\beta'(i-1))p^{\mathrm{h}(\beta'(i-1))}P_{i-1}'(\beta'(i-1),\alpha')\end{aligned}$$

4) **Calculation of (Maximum Average of) Differential Probability**
Evaluate $P_i(\beta(i),\alpha)$ by

$$P_i(\beta(i),\alpha)=p^{\mathrm{h}(\beta'(i))}P_i'(\beta'(i),\alpha')$$

With this procedure we can compute the truncated differential probability of PSN-structures and (the maximum average of) the differential probability of SPN-structures with 16 input words. Furthermore, applying this procedure to the each MixColumn transformations of Rijndael, allows us to compute the truncated differential probability and (the maximum average of) the differential probability. From this computation, the maximum average of the differential probability of 7-layer Rijndael including 4 nonlinear layers, i.e. 4-round Rijndael, is upper-bounded by $1.00*p^{16}$ ($=1.065*2^{-128}$) and that of 9-layer Rijndael including 5 nonlinear layers, i.e. 5-round Rijndael, is upper-bounded by $0.940*p^{16}$ ($=1.0007*2^{-128}$) [1]. To be secure against differential and truncated differential cryptanalysis, 2 more layers (1 round) are necessary to avoid the exhaustive search of the the last 2 layers (1 round). This implies a total of 80 S-boxes is needed.

[1] [12] stated that 5-round differential with probability $1.06*2^{-128}$ was found, but this was typo. The correct round is 4.

4.3 Truncated Differential Probability of E2′-like Transformation

Using the values of the differential probability of the $2r$-layer PSN-structures, we can calculate the truncated differential probability of E2′-like transformations recursively. In this calculation, we assume the random output-differential Feistel cipher, hence the probabilities for $\chi(\Delta x \oplus \Delta y) = 1$ and $\chi(\Delta x \oplus \Delta y) = 0$ for two random output-differential values $\Delta x, \Delta y, \in GF(2^n) \setminus \{0\}$ are $(2^n - 2)/(2^n - 1)$ and $1/(2^n - 1)$, respectively.

The procedure for calculating the truncated differential probability of the E2′-like transformation is as follows.

1) **Computation of Truncated Differential Probability of Round Functions**
Using the procedure for calculating truncated differential probability of $2r$-layer PSN-structure, calculate the truncated differential of round functions. Hereafter, we denote the truncated differential probability of the i-th round function for the truncated differential $(\zeta'(i), \zeta'(i-1))$ by $Q'_r(\zeta'(i), \zeta'(i-1)) = P(\chi(\Delta X(i)) = \zeta'(i) | \chi(\Delta X(i-1)) = \zeta'(i-1))$ for $\zeta'(i), \zeta'(i-1) \in GF(2)^m$

2) **Initialization**
Let $\zeta'(0) = (\Delta L'(0), \Delta R'(0)) \in GF(2)^{2m}$. For given $\eta' \in GF(2)^{2m}$, calculate $P'_0(\zeta'(0), \eta')$ for every $\zeta'(0) \in GF(2)^{2m}$, where we assume

$$P'_0(\zeta'(0), \eta') = \begin{cases} 1 & \text{if } \zeta'(0) = \eta' \\ 0 & \text{otherwise,} \end{cases}$$

3) **Recursive Computation of Truncated Differential Probability**
Let $\zeta'(i) = (\Delta L'(i), \Delta R'(i)) \in GF(2)^{2m}$, $\zeta(i) = (\Delta L(i), \Delta R(i)) \in GF(2)^{2m}$, where $\chi(\zeta(i)) = \zeta'(i), \chi(\Delta L(i)) = \Delta L'(i), \chi(\Delta R(i)) = \Delta R'(i)$. Utilizing the values of truncated differential probabilities of round functions, calculate $P'_i(\zeta'(i), \eta')$ recursively for every $\zeta'(i) \in GF(2)^{2m}$.

$$P'_i(\zeta'(i), \eta') = \sum_{\substack{\xi', \\ \chi(L(i-1) \oplus \xi) = \Delta R'(i), \\ \chi(\xi) = \xi'}} Q'_i(\xi', \Delta R'(i-1)) P'_{i-1}(\zeta'(i-1), \eta')$$

4) **Calculation of (Maximum Average of) Differential Probability**
Calculate $P_i(\zeta(i), \eta)$ by

$$P_i(\zeta(i), \eta) = p^{\mathrm{h}(\zeta'(i))} P'_i(\zeta'(i), \eta'),$$

where $\chi(\eta) = \eta'$.

4.4 (Maximum Average of) Differential and Truncated Differential Probability of E2′-like Transformation

In this subsection, we evaluate the maximum average of the differential probability of E2′-like transformations with proper initial transformations, where we assume the all linear layers are same as that of E2.

First we consider E2′-like transformations with 2-layer PSN-structures. In this case, a nonlinear layer with 16 nonlinear functions, or 2-round E2′-like transformations with 2-layer PSN-structures can be adopted as the approximately random output-differential initial transformation. 8-round E2′-like transformation with 2-layer PSN-structures with proper initial transformation has maximum average of differential probability of less than $0.940 * p^{16}$ $(= 1.0007 * 2^{-128})$. In this case, it is provably secure with 80 nonlinear functions. To offer security against differential cryptanalysis, 2 more rounds are necessary, which means it needs a total of 96 nonlinear functions.

If we slightly change linear transformation of SPN-structures, it can be provably secure with 72 nonlinear functions. To offer security against differential cryptanalysis, 2 more rounds are necessary, which means it needs a total of 88 nonlinear functions.

Next we consider E2′-like transformations with 4-layer PSN-structures. In this case, a nonlinear layer with 16 nonlinear functions or 2-round E2′-like transformations with 2-layer or 4-layer PSN-structures can be adopted as the proper initial transformation. A 5-round E2′-like transformation with 4-layer PSN-structures with proper initial transformation has maximum average of differential probability lower than $0.940 * p^{16}$ $(= 1.0007 * 2^{-128})$. In this case, it is provably secure with 96 nonlinear functions. To be secure against differential cryptanalysis, 1 more round is necessary, which means it needs a total of 112 nonlinear functions to avoid the exhaustive search of the final round.

On the other hand, an 8-round E2-like transformation with 3-layer SPN-structures, has maximum average of differential probability lower than $0.940 * p^{16}$ $(= 1.0007 * 2^{-128})$. In this case, it is provably secure with 128 S-boxes (in this case, approximately random output-differential initial function is not necessary because of the first nonlinear layers of the first and second S-boxes). To be secure against differential cryptanalysis, 1 more round is necessary, considering the exhaustive search of the final round, which implies it needs a total of 144 S-boxes.

These results means that E2′-like transformations with 2-layer PSN-structures is more secure than 3 or 4 layer.

The block cipher MISTY with 16-input words and 3-rounds has maximum average of differential probability equal to $p_{\max}^{16}$, where $p_{\max}$ is the maximum average of differential probability of nonlinear functions. In this case, it is provably secure with 81 S-boxes. To be secure against differential cryptanalysis, 1 more round is necessary, which implies it needs a total of 108 S-boxes.

4.5 Impossible Differential Cryptanalysis of Rijndael, E2′-like Transformation

Impossible differential cryptanalysis is a cryptanalysis against block ciphers which utilizes the pair of input and output-differentials whose differential probability equals 0 [1].

In the previous procedure, we proposed the procedure which calculates the truncated differential probability of random output-differential (Feistel) ciphers. It follows that from the relations between truncated differential probability and differential probability we can also calculate the differential probability.

From the values of the differential probability, our procedure can calculate the resistance against impossible differential cryptanalysis, by counting the number of differentials whose probabilities equal 0. In the case of E2′-like transformations with 2-layer PSN-structures, it can be proved that 9-rounds offer security against impossible differential cryptanalysis while 8-rounds do not. In the case of E2-like transformations with 3-layer SPN-structures, 8-rounds offer security against impossible differential cryptanalysis and 7-rounds do not. Comparing the numbers of nonlinear functions, E2′-like transformations with 2-layer PSN-structures is superior to E2-like transformations with 3-layer SPN-structures, i.e. the basic structure of block cipher E2.

In the case of Rijndael, it can be proved that 7-layers (including 4 nonlinear layers) offers security against impossible differential cryptanalysis while 5-layers (including 3 nonlinear layers) do not. Comparing the numbers of nonlinear functions, basic structure of Rijndael has a little higher level of security against impossible cryptanalysis than E2′-like transformation with 2-layer PSN-structures. However, considering the amount of linear layer operations, E2′-like transformations with 2-layer PSN-structures may be superior to the basic structure of Rijndael, because the linear layer of E2′-like transformations consists of only “xor” whereas that of Rijndael consists of heavier linear transformation over Galois field GF(2^8).

5 Conclusion

This paper examined the truncated differential probability and the differential probability of the word-oriented Markov ciphers and random output-differential (Feistel) ciphers like Rijndael and (modified) E2 and clarified the relations among the differential, truncated differential and the impossible differential cryptanalysis of the random output-differential (Feistel) cipher. This random output-differential (Feistel) cipher uses a weaker assumption than the assumption that all S-box differentials are equally likely. This is not a strong assumption because denying this model is equivalent to denying the truncated differential cryptanalysis. We then described a procedure for calculating the truncated differential probability and (maximum average of) the differential probability of such ciphers. Using this procedure, we computed and proved the security of Rijndael, E2 and the E2′-like transformation against differential, truncated differential and impossible differential cryptanalyses under the assumption of a random output-differential (Feistel) cipher. Our evaluation finds that Rijndael is the most secure, and the E2′-like transformation with 2-layer PSN structure is a little less secure. However, the linear transformation in E2′-like transformations is lighter than that of Rijndael and can be improved by slightly changing, so the overall speed may be the highest (may be "not" the highest). Our results implies that SPN-structures (like Rijndael, Serpent) and Feistel structures with S-boxes composed of 2-layer PSN-structures (like E2-like transformation with 2-layer PSN-structures) have no disadvantage in terms of security against differential and truncated differential cryptanalysis. We can similary evaluate the security of Feistel structures with S-boxes composed of 2-layer SPN-structures (like Twofish [18]) against differential and truncated differential cryptanalysis, though we have not evaluated Twofish yet because Twofish is not composed of just word-wise operations of fixed size. However, Feistel structures with 2-layer SPN-structures can be proved to be secure and have no disadvantage in terms of security against differential and truncated differential cryptanalysis, if we select the proper linear transformations in their SPN-structures.

References

[1] E.Biham, A.Biryukov and A.Shamir, "Cryptanalysis of Skipjack Reduced to 32 Rounds Using Impossible Differentials." J. Stern (Ed.): EUROCRYPT'99, LNCS 1592, pp. 12-23, Springer-Verlag, Berlin, 1999.

[2] E.Biham and A.Shamir. "Differential Cryptanalysis of DES-like Cryptosystems." Journal of Cryptology, Vol.4, No.1, pp.3-72, 1991. (The extended abstract was presented at CRYPTO'90).

[3] J. Daemen and V. Rijmen. "AES Proposal Rijndael," AES Round 1 Technical Evaluation CD-1: Documentation, National Institute of Standards and Technology, Aug 1998. See **http://www.nist.gov/aes**.

[4] T.Jakobsen and L.R.Knudsen. "The Interpolation Attack on Block Cipher." In E.Biham, editor, Fast Software Encryption - 4th International Workshop, FSE'97, Volume 1267 of Lecture Notes in Computer Science, pp.28-40, Berlin, Heidelberg, NewYork, Springer-Verlag, 1997.

[5] L.R. Knudsen and T.A. Berson, "Truncated Differentials of SAFER." In Fast Software Encryption - Third International Workshop, FSE'96, Volume 1039 of Lecture Notes in Computer Science, Berlin, Heidelberg, NewYork, Springer-Verlag, 1996.

[6] M. Kanda et al. "A New 128-bit Block Cipher E2" Technical Report of IEICE. ISEC98-12.

[7] X. Lai, J. L. Massey and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," Advances in Cryptography-EUROCRYPTO '91. Lecture Notes in Computer Science, Vol. 576. Springer-Verlag, Berlin, 1992, pages. 86-100.

[8] M.Matusi, "Linear Cryptanalysis Method for DES Cipher." In T. Helleseth, editor, Advances in Cryptology - EUROCRYPT'93, Volume765 of Lecture Notes in Computer

Science, pp.386-397. Springer-Verlag, Berlin, Heidelberg, NewYork, 1994. (A preliminary version written in Japanese was presented at SCIS93-3C).

[9] M. Matsui, “New structure of block ciphers with provable security against differential and linear cryptanalysis,” In Dieter Grollman, editor, Fast Software Encryption: Third International Workshop, volume 1039 of Lecture Notes in Computer Science, pages 205-218, Cambridge, UK, 21-23 February 1996. Springer-Verlag.

[10] M. Matsui, “New block encryption algorithm MISTY.” In Eli Biham, editor, Fast Software Encryption: 4th International Workshop, volume 1267 of Lecture Notes in Computer Science, pages 54-68, Haifa, Israel, 20-22 January 1997. Springer-Verlag

[11] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, “Handbook of Applied Cryptography”, CRC Press, 250-250 (1997).

[12] Moriai, S., Sugita, M., Aoki, K., Kanda, M. “Security of E2 against truncated Differential Cryptanalysis” Sixth Annual Workshop on Selected Areas in Cryptography (SAC’99), pages, 133-143 (1999).

[13] Moriai, S. et al. “Security of E2 against truncated Differential Cryptanalysis” Technical Report of IEICE, to appear.

[14] Matsui, M. and Tokita, T. “Cryptanalysis of a Reduced Version of the Block Cipher E2” in 6-th international workshop, preproceedings FSE’99

[15] K. Nyberg and L. R. Knudsen, “Provable security against a differential attack,” in Advances in Cryptology - EUROCRYTO’93, LNCS 765, pages 55-64, Springer-Verlag, Berlin, 1994.

[16] M. Sugita, “Security of Block Ciphers with SPN-Structures” Technical Report of IEICE. ISEC98-30.

[17] M. Sugita, K. Kobara, H. Imai, “Pseudorandomness and Maximum Average of Differential Probability of Block Ciphers with SPN-Structures like E2.” Second AES Workshop, 1999.

[18] B. Schneier, J. Kelsey, D. Whiting, D. Wagner and C. Hall, “Twofish: A 128-Bit Block Cipher,” AES Round 1 Technical Evaluation CD-1: Documentation, National Institute of Standards and Technology, June 1998. See **http://www.nist.gov/aes**.

[19] T. Tokita, M. Matsui, “On cryptanalysis of a byte-oriented cipher”, The 1999 Symposium on Cryptography and Information Security, pages 93-98 (In Japanese), Kobe, Japan, January 1999.

Session 6:

"AES Issues" Panel

AES and Future Resiliency: More Thoughts And Questions

By Don B. Johnson
djohnson@certicom.com
March 10, 2000

Introduction

In a paper submitted previously, the author argued that a new AES evaluation criterion of future resiliency be added, defined as the ability to respond to the uncertain future and that this criterion could best be met by NIST selecting multiple disparate AES winners. This paper continues that discussion by providing more rationale. It also asks some questions and explores possible outcomes of the AES process.

Summary From Previous Paper

The author's previous paper closed with this summary:
"NIST should carefully examine the various classification schemes that have been made and endeavor to choose the AES second round finalist candidates considering that it is a worthwhile goal to try to ensure that differing design approaches are included. This is because of reasons of future resiliency, extending cryptographic knowledge, Super AES, crypto toolbox philosophy, possible patent complications, target diffusion, avoidance of artificial tiebreakers, recognition of the problem being multidimensional with imperfect information, and the constraints of other standards organizations. That is, in selecting the handful of AES second round finalists, disparity of design approaches is to be desired over conformity."

For further explanation of these rationales, see the paper at the NIST AES website at www.nist.gov/aes. These rationales continue to be valid in the discussion regarding whether there should be multiple AES winners or not.

NIST is to be congratulated for their selection of AES finalists as they do represent a disparate selection from among the submitted AES candidates. Furthermore, the inventors of each finalist algorithm represent a significant portion of the skills in the cryptographic community. As the AES winner(s) must give a royalty-free license (if the algorithm is patented), perhaps the main rationale to participate in the AES process is the recognition one receives, with the winner(s) getting major "bragging rights." Another way to look at the NIST AES finalist selection is that NIST has put "five cats in a bag" to see who survives as each submitting group is highly motivated to find chinks in the armor of the other AES finalist algorithms. Better to find out now rather than later.

I) Additional Rationale for Multiple Algorithms

Space Probe Scenario

A reason to consider multiple winners is that sometimes one needs to use hardware for performance reasons, but the hardware is difficult or impossible to change once deployed. Consider a commercial space probe [JC]. Once it arrives at its destination, it must be essentially self-sufficient. Calling it back is out of the question. However, backup circuitry is a normal part of its design and this flexibility could be extended to include a backup symmetric cryptographic algorithm. As these types of projects might take years or decades, such an algorithm backup is simply prudent.

AES Selection Time

Another factor that should be considered by NIST is the amount of time that was taken by the AES process. If a sole AES winner were to prove unfortunate for some reason, then it could take many years to determine a substitute. It has been said that three months is considered an Internet year. The time needed to do another AES process may not meet the requirements of the market.

Infrastructure Overoptimization

As we saw with the deployment of DES, the selection of one algorithm by NIST meant that best-practices resulted in the use of that **one** algorithm. For much of the life of DES, there was no pressing need for vendors to try to design systems to support multiple symmetric cryptographic algorithms, DES was it. With DES the only choice, this simplified things for a vendor. However, we see today that this simplification resulted in a deployed infrastructure where there are concerns that some portions are now vulnerable to a determined attack.

Einstein is reputed to have said, “One should try to make things as simple as possible, but no simpler.” Even the selection by NIST of as few as two winners will mean that vendors will need to design in flexibility of algorithm choice in some products and provide for the possibility of algorithm replacement in others, rather than overoptimize as was done in the case of DES.

NIST as AES Architect

NIST is overseeing the AES process. As such, NIST is the architect of the AES process, that is, it is creating the AES design architecture. There are two fundamental responsibilities of an architect, as follows:

1) Specify **enough** detail to allow others to proceed.
2) Know what **not to specify** to allow creativity and flexibility in others.

In this AES architect role, NIST should follow the general principle of “If in doubt, don’t.” NIST/NSA can and should make “apparent health” statements on the security of the AES finalists. NIST can and should make decisions about which AES finalist algorithms are suitable for government use, using whatever additional criteria (if any) besides security that NIST deems appropriate. This is ALL that NIST should try to do. NIST should resist the temptation to try to solve potential challenges resulting from the existence of multiple algorithms, such as the need to negotiate algorithms or the need for a vendor or market segment to select the most appropriate algorithm.

NIST or the Marketplace?

Asking NIST to select a sole AES winner means that one believes this decision is appropriate for top-down decision-making, as in a command economy or an army. A top-down methodology is appropriate when any decision is better than no decision (e.g., traffic lights) or when a decision must be made quickly (e.g., a battle). However, simply as a matter of information flow, all the relevant information cannot be expected to be available to the responsible top-level decisionmaker. The marketplace (bottom-up decision-making) has been shown to be much more responsive and adaptable than a command economy. This is because each economic entity or group of entities makes decisions based on its own information and needs.

So one question that NIST needs to ask itself is does it see the AES process (that is, the development of commercially-appropriate symmetric cipher or ciphers) as needing a top-down decision to be made or does it believe that the marketplace is the most appropriate place for this decision to be made. The marketplace has a way of determining what is appropriate; if there is truly one finalist that is superior in many ways, it does not need NIST's selection of it as the winner to emerge as the winner in the marketplace. However, there is a real concern that NIST could make a suboptimal choice due to insufficient information. In this case, "hands off" is the wisest course of action.

NIST needs to resist the temptation to make a decision in an area beyond their (or anyone's) competence. The round 2 discussion issues asking questions about how to assess speed versus security margin, need for low-end flexibility, and hardware versus software performance indicate that NIST recognizes its lack of certainty in these areas. This is fundamentally because there are no obviously single correct answers to these questions. Different applications may require different answers to these questions. NIST should make a virtue of its (really, everyone's) ignorance and not attempt to decide these unanswerable questions one way or the other, but let others make each decision that is most appropriate for them.

Bias?

It may not be politically correct to say so, but NIST should understand that any counsel given it might be biased; this might be especially true of counsel from submitters of algorithms. This is not necessarily a bad thing, the submitters of the AES finalists have very high cryptographic skills and it is certain that the submitters made their decisions after thinking long and hard about the problem. It is just that each submitter naturally thinks their beliefs are correct.

For example, it would be no surprise that a designer of a very flexible algorithm might think that flexibility is an important AES criterion. That is likely one of the reasons the submitted algorithm was made flexible in the first place, so that it would have an advantage when compared with other AES candidates.

The point is that if NIST were to announce they are seeking a single winner then this (in turn) results in a ranking of finalists, just as identifying any other AES criterion as critically important would also potentially rank the finalists. However, note that if an algorithm is truly more flexible than another, it still stands a greater chance of being used in the "marketplace" selection process mentioned above. That is, any advantages of an algorithm remain advantages; by selecting multiple winners and relying on the marketplace, NIST is not required to try to determine which advantages are more important than others.

II) Some Questions

Quantum Computers

One big question regarding the future is whether or not quantum computers are feasible and if they are, what effect they will have on cryptography. An arbitrary bitsize quantum computer (assuming it can be built) allows a square root attack on a symmetric cipher. The possibility of this provides some justification for the larger AES keysizes; a 256-bit symmetric cipher would take 2**128 quantum operations to exhaust the key space.

However, an interesting question is whether there is some limit in practice to the number of bits of a quantum computer. Many researchers suspect this is the case, that quantum decoherence will prove insurmountable for some number of quantum bits.

In terms of AES this question becomes: if one can only build an x-bit quantum computer, how much does this help in attacking each AES finalist? As all block ciphers are composed of smaller chunks, how might these chunks interact with a quantum computer? This possibility can be termed a partial quantum attack. And of course, an adversary could construct many quantum computers to run the attack in parallel, assuming this would help. So the question is: "How does a parallel partial quantum computer affect the ability to attack the AES finalists?"

As an example, DES is composed of a 56-bit key. A 56-bit quantum computer should be able to attack the DES. However, the DES design is such that each of the sixteen rounds uses a 48-bit key. This suggests the possibility that a 48-bit quantum computer might somehow be able to be used to successfully attack the DES. The question of how the AES finalists stack up in relation to parallel partial quantum computers is a critical question to be answered. NIST should step up to this analysis if it is not forthcoming from the research community. No final AES decision should be made without some exploration of the expected effects of this possibility.

Random Cipher

It is clear that a random cipher for a certain blocksize is the unrealizable ideal. This is a cipher that selects a random choice for the output block for each input block, the key providing an index into a set of random selections. There is no structure that is able to be attacked by an adversary. The best attack is key exhaustion, which is the goal of any symmetric cipher. It is also clear that such a ideal block cipher is totally impractical as the space needed is totally infeasible. However, one would like any particular block cipher to "appear" to be ideal to an adversary. That is, even though the structure is known to an adversary, this structure does not allow any shortcuts to be made. A critical question is whether a AES finalist appears "random." There are many established randomness tests. Any deviation from random is a cause for concern.

Another related important question is at what point do degenerate forms of a finalist not appear random. For example, a finalist may have 20 rounds. It is important to know if the output after 4 rounds appears random or if it takes 8 or even 16 rounds. This is important as it gives an indication of the margin of safety built into the cipher. It is obvious that a round of cipher A cannot be considered equivalent to a round of cipher B but this type of analysis allows one to at least map some internals of one algorithm to another for comparison purposes.

Knowing what to do with this analysis is more problematical. Regardless, this is an important data point. If I know that cipher A is essentially as fast as cipher B, but that cipher A results in random-appearing output after 5 of 16 rounds and cipher B results in random-appearing output after 8 of 12 rounds, then cipher A may be the more conservative choice in some sense. But NIST should be wary of this analysis, one can simply add more rounds at a performance cost.

Should a cipher be rewarded (or penalized) for minimizing overhead? Should a cipher be rewarded (or penalized) if it has “more” rounds? This means that (apparent) security and performance are very closely tied together.

Combined Attacks

In the real world, the adversary is able to combine the effects of various attacks. Even if each attack results in only a relatively small advantage that is not relevant when considered by itself, a combination of attacks may accumulate to result in a feasible attack. For this reason, any discovered theoretical advantage for an adversary attacking an AES finalist (no matter how apparently small) is a concern.

III) Thoughts on the AES Finalists

Following are some thoughts on the AES finalists. It should be recognized that these ideas are tentative and subject to improvement and correction. Of course, the detection of any security flaw in a finalist would have a major impact. Each finalist algorithm can be seen as a statement by the designers regarding not only one way to solve the various tradeoffs of the AES puzzle, but also as how the designers see the future. It is hoped that these thoughts on the finalists are used by NIST in the spirit in which they are given, as food for thought.

MARS

MARS was designed with some thought to try to avoid potential future attacks, especially in its heterogeneous structure, a keyed-core surrounded by unkeyed forwards- and backwards-mixing functions. The unkeyed mixing functions cost time and space, but their inclusion seemed prudent to the designers and worth the cost. The core “mixing” function uses addition, multiplication, fixed and data-dependent rotations, and an S-Box (straightforward substitution cipher). The designers responded to criticism to improve the performance of MARS by using the “tweak” allowed by NIST.

From a perspective on the future, the designers of MARS believed the best way to handle uncertainty was to use many different techniques using a cost/benefit analysis. The MARS design is the most different of the Feistel cipher finalists. Another way to look at MARS is that IBM is a large organization which had many people with good ideas trying to get them incorporated into the IBM submission. This can be seen in the number of authors of the MARS paper.

From a perspective of future resiliency, the inventors of MARS thought that a heterogeneous structure was important.

RC6

RC6 was built from a heritage of RC5 and was designed to be fast and simple to describe. The core ideas of RC6 came from RC5, which was designed by one person, as such it represents a unity of design approach. In many scenarios, RC6 is the fastest AES finalist. The pseudocode for RC6 is very straightforward with basic operations defined on 32-bit words; the RC6 pseudocode is the shortest of all finalists. It uses addition, multiplication, data-dependent rotations and substitution to do the cryptographic "mixing." RC6 can be seen as an example of building a performance-optimized cipher on the idea of data-dependent rotations. The challenge for the designers of RC6 is to show that their design is not **too simple**. For example, comparing RC6 to MARS, MARS adds more complexity to its specification to try to provide more mixing.

Indeed, the "Correlations in RC6" paper by Knudsen and Meier (available at www.nist.gov/aes) indicate that reduced rounds of RC6 do not appear random. The observation by Saarinen in the NIST RC6 forum on finding "almost equivalent" keys in RC6 suggests other possible concerns. These ideas hint that RC6 may be on the edge of security.

From a future resiliency perspective, the designers of RC6 believed that parameterization was paramount. In this way, if a certain number of rounds was found to be weak, this number could be adjusted upwards.

Rijndael

Rijndael does not use a Feistel structure, rather it uses a matrix structure where the cryptographic mixing involves byte substitution, row shifting and column multiplication. Rijndael has the most different structure when compared with the other AES finalists. It can be implemented using byte operations and is therefore very flexible.

From a future resiliency perspective, the designers of Rijndael were willing to go in new directions and wanted high flexibility in implementation.

Serpent

Serpent is a conservative design and deliberately tries to build on the vast amount of information relating to DES. Serpent is also the **slowest** of the five AES finalists on most platforms. Being the slowest, the challenge for the designers of Serpent is to try to show how the other finalist algorithms cut corners in ways that Serpent did not (that is, the additional performance cost should be justified). For example, suppose that NIST gave all five AES finalists "certificates of apparent security," it is not clear what symmetric algorithm niche would best be filled by use of Serpent, as opposed to one of the other finalists. Of course, a specific implementation might find that Serpent is the fastest method, if the instructions it uses are fast and the instructions that other methods use are slow.

The designer's of Serpent have presented an "equivalent rounds" analysis of the AES candidates and tried to show how Serpent uses more rounds than might be thought needed as a safety margin. Yet the designers did not officially change the specification of Serpent (even though they knew that there were many other faster AES candidates) so they must believe they have good reasons for designing it as they did. Serpent and RC6 appear to have opposite design philosophies in this area of tradeoff between security margin and performance.

From a future resiliency perspective, the designers of Serpent decided to use more rounds and affect performance to try to achieve a higher security margin of safety. This means Serpent may have some performance concerns, at least when compared with the alternatives.

Twofish

Twofish is a byte-oriented Feistel cipher with great flexibility of implementation, allowing a wide range of time/space tradeoffs. Many research reports have been written on various aspects of Twofish, which give confidence in its security. There was also a cost/benefit analysis done by the designers to decide which operations to use.

From a future resiliency perspective, Twofish's goals were security and implementation flexibility.

IV) Possible Outcomes

Does NIST want the fastest cipher? ... the cipher with the largest safety margin? ... the cipher with the most flexibility? ... the cipher with the most disparate instructions? ... the most Feistel-like cipher? ... the cipher with the most disparate design? Single or multiple winners? ... some other criteria? The point is that different answers to each question can lead to a different ordering of the AES finalists. Furthermore, any selection by NIST indicates in a backwards fashion which criteria they decided was more important than others. As one example, comparing MARS and Serpent, are more rounds or different rounds the better way to address having a sufficient safety margin? As another example, comparing MARS and RC6, are many different ideas or unity of design the better way to design a cipher?

The problem for NIST is not that there are no answers, it is that there are **too many** rational answers. Barring a security flaw, any of the AES finalists could be justified as being the sole winner simply by NIST adopting the corresponding design philosophy behind the winner as its own. NIST should resist any temptation to do this. Rather, as each submission has a different design philosophy, NIST should accept the implication that there was no obvious single all-around best solution. NIST should accept this implicit "higher-level" statement from the submitters and agree with them (as a group) that there is no single all-around best answer.

Strictly speaking, NIST's AES mandate is to select a winner or winners that is/are suitable for use by the US Federal government to protect sensitive non-classified data. Following the historical pattern of DES, it is also expected that NIST/NSA will issue a statement that the winner(s) is/are suitable for the intended purpose. Historically, it was this endorsement that gave confidence to other groups, such as the American Bankers Association, to also endorse DES, which in turn led to DES becoming the most-deployed commercial cryptographic algorithm.

Now, some 25 years after DES, we see the endorsement by NIST of 3 families of asymmetric cryptographic algorithms in the revision of FIPS 186; namely, those based on the difficulty of integer factorization, the normal discrete logarithm, and the elliptic curve discrete logarithm. This allows the advantages of each method to determine the way asymmetric cryptography rolls out in the future. That is, NIST recognizes that there are multiple answers to the asymmetric cryptography question.

This author hopes that similar rationale will prevail among the NIST AES selection team regarding the symmetric cryptography question. While this author believes that the best outcome of the AES process is a handful of winners which lets the marketplace determine each algorithm’s niche, it is realized that not all others share this opinion.

Ranking?

NIST should realize its decision is not restricted between having one AES winner and having multiple winners, it could also decide to have a ranking among multiple winners. As an example, NIST might specify that algorithm A is the primary winner and algorithm B is the backup. In this example, an implementation would be expected to either implement algorithm A (if resources are constrained) or both algorithms A and B (if resources are available). This seems much preferable to declaring a single AES winner, although inferior to selecting multiple co-equal winners.

Multiple Endorsement?

Another alternative is that regardless whether one or multiple winners (ranked or not) are selected by NIST for use by the US Federal government, NIST/NSA could issue health statements that certain finalists meet their intended security goals. This would at least allow other standards bodies to negotiate with increased confidence for the rights to an endorsed algorithm, if that algorithm better met their needs. For example, NIST might say that algorithm A wins (for US

Federal government use), but also issue a NIST/NSA report that algorithms A, B, and C meet their intended security goals.

Just to be clear on this point, if all five AES finalists have no known security weaknesses, then all five finalists should be giving a "certificate of health" regardless of the decision regarding the number or specific selection of AES winner(s) for approval for US Federal government use.

Acknowledgements

The author wishes to thank Certicom for providing the environment in which to write this paper.

Reference

[JC] Jerry Coffin in a post on sci.crypt on AES mentioned that satellites were an example where hardware was infeasible to change once deployed and saw this as a reason to have multiple winners.

Biography

Don B. Johnson is Director of Cryptographic Standards for Certicom, is a member of Certicom Research, and sits on the Advisory Board of the Standards for Efficient Cryptography Group (SECG). He participates in ISO SC27, ANSI X9, IEEE P1363 and other standards bodies. He has over 40 patents and patent applications in the area of cryptography. He was the editor of the X9.62 Elliptic Curve Digital Signature Algorithm (ECDSA) standard.

The Effects of Multiple Algorithms in the Advanced Encryption Standard

Ian Harvey, nCipher Corporation Ltd, 4th January 2000

Abstract

This paper presents a discussion of the issues relating to the selection of encryption algorithms in practical situations. An AES standard which recommends multiple algorithms in a variety of ways is discussed, and it is shown that this can present an overall advantage.

Introduction

The Advanced Encryption Standard aims to become the first choice for most situations requiring a block cipher. To do this it has to satisfy a wide variety of requirements for - amongst other things - security, performance, and resource constraints.

Each of the current five candidate algorithms for AES satisfies a different balance of these constraints; the 'best' algorithm depends on circumstances, which are impossible to know beforehand. Furthermore, one of the principal requirements - that of security of the algorithm - cannot easily be measured; subjective judgements therefore must be made (based, for instance, on notions of 'safety margin' or 'conservative design') which may prove to be inaccurate or irrelevant in years to come.

In the absence of a precise definition of what constitutes the 'best' algorithm, or accurate means even to measure this, any choice of algorithm is somewhat arbitrary. In security terms this seems needlessly risky, and it has been suggested that avoiding the need for a single final algorithm would have advantages.

In the sections below, we discuss the factors which affect algorithm choice, and then examine the effect on these of an AES which offers multiple algorithms.

Factors in algorithm choice

Many factors may be involved in the selection of an algorithm. In most cases, one single factor is overwhelmingly the most important, and often some are of little or no importance.

Aside from performance issues, which are discussed later, criteria for algorithm selection include:

- **Security against theoretical attacks**

The reputation of an algorithm is frequently a major factor in its selection, both in terms of the design of the algorithm, and the extent to which it has been studied for cryptanalysis. A theoretical attack does not have to become practical before the cipher is rendered commercially unusable (for instance, liability insurance on a system using it may become void).

It is to be expected that no candidate with a known theoretical weakness will be given final recommendation within AES. Furthermore, the effect of the AES 'brand name' will be to concentrate research into the selected algorithms; this will (all being well) improve their reputation as time passes.

- **Security of implementations**

Some of the most practicable attacks of recent years have been directed against particular implementations of algorithms, rather than their theoretical definitions. These include timing attacks, power-analysis attacks, and fault-induction attacks.

In general, defending against these attacks is done when the implementation is designed, using a range of proprietary techniques. Some algorithms may have features which make them particularly difficult to defend, but it is generally not possible simply to define a 'good' feature set. Selection of an algorithm to resist implementation attacks can often only be done when the threat model has been decided, and little generalised guidance can be given.

- **Cost of implementation**

The effort required to correctly implement a given algorithm is frequently a major issue. For software implementations, the factors which affect this include:
- availability of reference implementations in a given language
- ease of understanding and adapting the reference implementation
- complexity of any optimisations required for optimum performance, and
- the ease and completeness of correctness testing.

For hardware implementation, important factors are:
- the complexity of the cipher
- the clarity of the cipher's given description
- availability of test vectors sufficient to give complete coverage.

Many developers will want to choose AES on the grounds of easy access to good reference material.

- **Architectural implications**

The precise functional 'shape' of an algorithm will have an impact on the way systems and protocols are designed to accommodate the algorithm. The block size and key size are the principal parameters, and fortunately all AES candidates are required to be compatible here.

However, additional parameters or features offered by particular algorithms - variable number of rounds, keys of other than standard length - may create additional complexities. Where it is not possible to adapt existing protocols to deal with these parameters, behaviour is often implied. This can be a major cause of interoperability problems - something which must not be allowed to bedevil the AES.

- **Legal issues**

Patent, copyright, and export-control issues affect no area of computing more than cryptography. Commercial developers will accept some licensing costs, but only up to a point - many of the potentially superior alternatives to DES (e.g. IDEA or RC5) have not been deployed widely, mainly for cost reasons. Free software developers are often unable to accept any restrictions on algorithm use.

One of the goals of the AES process is to produce a cipher which can be deployed universally, and a leading reason for choosing it will be freedom from legal impediments.

Performance issues

Every situation has its own unique performance criteria, which are invariably a trade-off between system requirements and speed, given 'enough' security. There are three main categories:

- **Best ideal-case speed**

The highest bit rate is required, irrespective of implementation complexity. The platform for deployment can be chosen (or at least unsuitable ones eliminated). Typically this means an algorithm which does well when hand-optimised in assembler for a modern processor, or can use parallelism in a large ASIC.

This type of performance is required by high-end hardware manufacturers, software developers who choose to target few platforms, and users who can choose platforms for best performance.

- **Best worst-case speed**

An acceptable bit rate is required, on a wide variety of platforms, or on a relatively non-standard platform. There should be no platforms on which speed is significantly lower than an alternative algorithm.

This type of performance is required by software developers who target a broad range of platforms, and is often associated with good speed available from a portable C implementation. It is also of significance to manufacturers (of e.g. embedded systems) whose choice of platform determined by other factors and cryptography is a secondary consideration.

- **Minimum implementation size**

Bit rate is not important, but constraints are placed on the resources required: gate count, code size or table size.

This type of performance is required by manufacturers of embedded systems for mass deployment, where unit hardware cost is critical. However, this group typically has much less need of interoperability outside the embedded application. The main reason to choose AES in this case will be the brand-name security.

It should be noted that performance and available resources will increase dramatically over time, but resistance to attacks will decrease. A standard intended for the long term should favour security over performance or resource requirements.

Approaches towards multiple algorithms

An Advanced Encryption Standard may be proposed which recommends more than one algorithm. There are a number of ways in which this might be done.

Multiple algorithms may be made optional; they must however be specified in such a way that any conforming AES implementation can interoperate with any other.

In some situations, both encryption and decryption can be controlled by one party, but in others they are controlled by separate parties and the choice of algorithm must be a mutually acceptable one. Any two such implementations must therefore share an algorithm, and the AES recommendations must guarantee this. The following approaches will be suggested:

A. All AES implementations must include all algorithms.

B. All AES implementations must include one primary algorithm, and a choice of secondary algorithms (possibly also ranked in order). Implementations will include secondary algorithms if it is to their advantage.

C. Given a set of N algorithms, an AES implementation must include at least N/2+1 algorithms from that set - this ensures any two implementations have at least one algorithm in common. Implementations will choose the subset of algorithms that best fulfils their requirements.

Theoretical Security / Implementation Security

Properly managed, multiple algorithm choice should enhance security. Should one algorithm fall to cryptanalysis, a second choice will already be available to provide backup. Also, given a choice, an developer will be able to pick the algorithm which best resists implementation attacks in the available technology.

Improperly managed, multiple algorithm choice will detract from security. If the choice of algorithm can be subverted in a given protocol, an attacker will be able to pick the easiest target.

Approach A is the most robust; two communicating parties can negotiate the 'strongest' algorithm and it will be used. Should an algorithm be broken, it is simply removed and the next best is selected.

Approach B allows implementers to choose to implement the secondary algorithms if they require a fallback. If secondary algorithms are broken first, all systems can revert to the primary algorithm, but if this is broken, some systems may not have an alternative. This scheme is therefore as resilient as approach A, except where systems omit the secondary algorithms due to cost considerations. In this approach, it would be prudent to choose a primary algorithm with a good security 'safety margin'.

Approach C, would in theory allow implementers to implement their choice of the 'strongest' algorithms, and two communicating parties would agree on at least one they considered secure. However, should *any* algorithm subsequently be broken, some combinations of communicating parties will be left unable to communicate. Also, this relies on the implementers holding opinions about algorithm security, which is contrary to the spirit of a security standard.

In practice, this makes approach C less secure than a single-algorithm selection.

The beneficial effect of an Advanced Encryption Standard concentrating cryptanalytic efforts will be diluted if too many algorithms are chosen. Approach B will present a primary target for research, and is best in this regard.

Cost of implementation

Any approach that mandates more than one algorithm to be implemented will raise the development costs proportionally. Approach A particularly, and to a lesser extent C, have the most impact. Where development costs are the overriding concern, approach B is as good as a single-algorithm selection.

The cost of implementation can be reduced dramatically, however, given good reference materials to accompany the standard. It is to be hoped that the various software implementations made available during the selection process will also be available to accompany the standard itself. This will strongly reduce the cost of producing a correct implementation.

It may be necessary to rewrite the descriptions of one or more algorithms to use a consistent set of terminology - particularly, for instance, with respect to bit-numbering and byte-ordering conventions.

Architectural implications

Multiple algorithms, and the process required to select one, will undoubtedly add to the architectural changes required for the new standard. In many situations, where a negotiation of cipher suite is already part of the protocol, this will have minimal additional impact.

Approaches A and B allow the selection to be made fixed, but approach C necessitates some form of negotiation, and this may be impossible in some circumstances.

The issue of additional algorithm parameters needs careful consideration. The table below gives some potential variation in parameters for each of the current AES candidates:

Cipher	Variations
MARS	Key size 4-39 32-bit words
RC6	Word size *w* , no. of rounds *r*, key size 0-255 bytes
Rijndael	Block length of 128, 192 or 256 bits
Serpent	Key size 0..256 bits
Twofish	Key size 0..32 bytes

All block ciphers can accommodate a 128-bit block, and 128, 192 and 256-bit keys as per the AES requirements, but beyond this the functionality differs substantially. In fact no two candidates have exactly the same set of allowed keys. It is poor software engineering practice to expose this to the user of the cipher; any variant in algorithm should *exactly* match the functional interface of the other, including rejecting the same set of invalid keys.

Similarly, if any variations in the number of rounds is proposed to allow a speed/security trade-off to be chosen by the user, it is poor design to let the user choose the number of rounds directly. Apart from the dangerous possibility of a round-by-round attack, it requires the user to know 'good' and 'bad' values for each algorithm. An acceptable solution would be to allow, say, three security levels - 'minimum', 'medium', and 'maximum', which is translated to a number of rounds appropriate to the algorithm in use.

Any AES which recommends more than one algorithm must address these issues, to remove ambiguity and promote interoperability.

Legal Issues

Clearly, multiple algorithms may increase the legal complications for a developer. In an ideal case, *any* algorithm offered as an option in the final AES will be free for use without restriction. As a minimum, sufficient algorithms should be available to construct a conforming implementation, without any patent or similar restriction.

Approach A requires all algorithms to have no restrictions; approach C requires the majority to have no restrictions, and approach B requires the primary algorithm alone to have no restrictions.

Performance - ideal case

Multiple algorithms give the best opportunity to maximise absolute speed, especially as evolving technology changes the balance between operations in different algorithms.

Approach A is good for this; the communicating parties will negotiate the fastest algorithm, and this is guaranteed to be available.

Approach B has some merit; the primary algorithm may not be the fastest on the chosen platform, but the implementer can add the secondary algorithms if they improve speed. In the ideal case, both communicating parties will do this and speed is maximised.

Approach C also allows implementers to choose the subset of algorithms which are most efficient on the chosen platform. Two communicating parties should then be able to pick their mutually fastest choice.

Performance - best worst-case

This benefits greatly from a choice of algorithms. Most worst-cases will be a particular feature of an algorithm which behaves poorly on a particular platform, and often any alternative will help.

The same strategy for choosing algorithms can be adopted as in the 'ideal case' scenario, with similar results. The worst case in Approach B is when the primary algorithm has poor performance on a given platform, and one of the communicating parties does not support any secondary algorithms. This is still an improvement, however, because it will only occur in those few cases where there is an overwhelming need for cost saving.

Minimum size requirements

This is impeded by the requirement for multiple algorithms; any standard which mandates more than one to be implemented will severely impact costs for low-end system developers.

To a certain extent this can be mitigated where two algorithms share common large functional blocks, or memory requirements which can be overlaid. To demonstrate this, the table below summarises the major functional requirements for each AES candidate. For comparison, triple-DES is also listed.

The table sizes given are ‘minimum’ requirements, with a 128-bit key, and will not be for the most efficient possible implementation; the RAM size given does not include that required for working purposes. The ROM size may be misleading as it does not include code size, which is in some cases traded off against lookup table size.

Candidate	ALU Operations					Table size (bytes)	
	logic / fixed shift	add/sub	data-dep shift	GF (2^P) ops	mult	ROM (S-boxes, etc.)	RAM (key schedule, etc.)
MARS	X	X	X		X	2048	160
RC6	X	X	X		X	none	176
Rijndael	X			X		512	16
Serpent	X					128	32
Twofish	X	X		X		64	24
Triple-DES	X					256	24

So it can be seen, for instance, that adding RC6 to a chip designed to implement MARS would have relatively little impact, but adding it to one optimised for Rijndael might be difficult.

Approach A is the worst of all worlds for minimum-size implementations. Approach C is better (only the smallest algorithms should be selected) but would still be typically double the best-case cost. Approach B allows very resource-limited implementations to implement solely the primary algorithm, and is as good as the single-algorithm case.

Summary of results

The effect of the various possibilities for a multiple-algorithm standard can be summarised in the table below, where “++” indicates the most positive impact, ‘0’ indicates no impact compared to a single-algorithm standard, and “--“ is the most negative impact.

Category	A	B	C	Notes
Security	++	+	--	1
Impl. Cost	--	0	-	2
Architecture	-	-	--	2
Legal issues	--	0	-	2
Best-case speed	++	++	++	
Worst-case speed	++	+	++	
Minimum size	--	0	-	

Notes:

1. Based on the ability of the standard to continue given failure of a cipher.
2. Can be mitigated by a good standardisation process.

Conclusions

A number of approaches to specifying multiple algorithms have been presented. This suggests that approach B - to specify a required 'primary' algorithm and one or more optional 'secondary' algorithms - has advantages over other approaches, and allows potential speed and security improvements over a single algorithm selection.

This approach means that outright performance can be eliminated from the criteria for primary algorithm selection - this can be left for the secondary algorithm. Security (or in practice, safety margin and conservative design) should be the primary algorithm's main requirement, followed by a modest resource requirement for minimum-size implementations.

Similarly, resource requirements can be ignored when making the secondary algorithm selection; implementers seeking a lowest-cost solution can simply omit these. An algorithm more aggressively optimised for performance is ideal here.

Provided that the legal and functional differences between the algorithms are mitigated by a well-written standard, there is no reason that this approach should not offer the best of all worlds.

Session 7:

"ASIC Evaluations / Individual Algorithm Testing"

Hardware Evaluation of the AES Finalists

Tetsuya ICHIKAWA* Tomomi KASUYA** Mitsuru MATSUI**

* Kamakura Office, Mitsubishi Electric Engineering Company Limited
ichikawa@harriet.mee-unet.ocn.ne.jp
** Information Technology R&D Center, Mitsubishi Electric Corporation
kasuya@iss.isl.melco.co.jp, matsui@iss.isl.melco.co.jp

1. Introduction

This report describes our evaluation results of implementing hardware of the AES finalists, concentrating on 128-bit key version, using Mitsubishi Electric's 0.35 micron CMOS ASIC design library. Our goal is to estimate the "critical path length" of data encryption /decryption logic and key setup time of key scheduling logic for each algorithm, which corresponds to the fastest possible encryption speed in feedback modes of operation such as CBC etc. To achieve this, we wrote fully loop-unrolled codes in Verilog-HDL language without introducing pipeline structure that blocks the feedback.

We first tried to investigate the evaluation environments to be used in NSA, especially the hardware design library, since NSA is expected to join the Round Two hardware analysis as has been shown in the NIST AES homepage [NIST (1998)]. However, after communicating NIST and MOSIS, we found that the library is an internal 0.5 micron standard cell library that is not available outside NSA, and a non-proprietary version of the library has not been developed. We therefore decided to analyze the AES finalists using Mitsubishi Electric's CMOS ASIC design library, whose information is publicly available in [MITSUBISHI (1997)].

Our simulation results show that Rjindael is the fastest as expected and it is even faster than DES, and Serpent is the next. Twofish, Mars and RC6 are slower than Triple-DES. We should note that since we used a general ECA (embedded cell array) library without applying special performance optimization techniques, these algorithms that heavily use arithmetic operations could be much faster if we introduce more expensive semi- or full-custom designs. However our analysis also indicates that even such designs are not expected to give a significant impact to change the ranking of the critical path length.

2. The AES Finalists

NIST announced the five AES finalists, in August 1999. This section briefly summarizes these algorithms, mainly data encryption operations, from hardware viewpoint.

2.1 Mars

Mars supports 128-bit blocks and a variable key size from 128 bits to 448 bits. It is designed to take advantage of the powerful operations supported on today's computers [Burwick et. al. (1999)].

The encryption part of Mars, which is composed of four kinds of round functions, is performed as follows. We have also listed major components that have an impact in hardware performance.

-The initial key addition
 4 additions mod 2^{32}.
-The unkeyed forward mixing (8 rounds)
 2 additions mod 2^{32}, and 4 look-up tables
 with 8bit-input/32bit-output.
-The keyed forward transformation (8 rounds)
 6 additions and 2 multiplications mod 2^{32},
 and 4 data-dependent rotations.
-The keyed backwards transformation (8 rounds)

6 additions and 2 multiplications mod 2^{32},
and 4 data-dependent rotations.
-The unkeyed backwards mixing (8 rounds)
2 subtractions mod 2^{32}, and 4 look-up
tables with 8bit-input/32bit-output.
-The final key addition
4 subtractions mod 2^{32}.

It seems that the heavy use of arithmetic operations, especially multiplications and additions mod 2^{32}, makes hardware slower and larger unless they are specially designed in a transistor level.

2.2 RC6

RC6 has three variable parameters, i.e., the number of rounds, the data block size, and the key size up to 2040 bits. The proposed version in AES has 20 rounds with a total of 4 additions (subtractions) mod 2^{32} before and after the round functions [Rivest (1998)], [RSA (1998)]. The major hardware components in the round function are as follows:

2 additions and 2 multiplications mod 2^{32},
2 data-dependent rotations.

These operations are well supported and fast on modern microprocessors, but expensive in hardware, especially multiplications and additions mod 2^{32}, make hardware slower and larger unless they are specially designed in a transistor level.

2.3 Rijndael

Rijndael also has a variable block length and a variable key length. The block length and the key length can be independently specified to 128, 192 or 256 bits. The proposed number of rounds in AES is 10, 12 and 14 when the key length is 128 bits, 192 bits and 256 bits, respectively [J.Daemen and V.Rijmen (1998)].
The round function of Rijndael in 128-bit blocks is composed of four distinct invertible transformations as follows:

-The ByteSub transformation
16 lookup tables with 8bit-input/output.
-The ShiftRow transformation
no hardware operations.
-The MixColumn transformation
logical AND and XOR operations.
-The AddRoundKey transformation
logical XOR operations.

Before the first round, the AddRoundKey transformation is also performed, and in the final round, the MixColumn transformation is omitted.
The basic components of Rijndael are logical operations and lookup tables; the latter is actually a composite function of an inversion over GF(2^8) with an affine mapping. Hence the structure of Rijndael is expected to be suitable for hardware implementation.

2.4 Serpent

Serpent has a 32-round SP-network structure with initial and final permutations, whose round function consists of 32 lookup tables with 4-bit input/output, logical and rotate shifts, and XOR operations [Anderson, Biham and Knudsen (1998)], [Biham (1997)].
These components are suitable for hardware implementation; particularly the small table size is expected to make hardware sufficiently small and fast.

2.5 Twofish

Twofish has a 16-round Feistel-like structure with an additional whitening of the input and output that consists of XOR operations. The major hardware components of the round function are as follows:

n lookup tables with 8-bit input/ output,
4 additions mod 2^{32},
logical AND and XOR operations,

The lookup tables can be also generated from another smaller 8 lookup tables with 4-bit input/output, and ***n*** is 12, 16 or 20 when the key length is 128, 192 and 256, respectively.
Twofish is not using particularly heavy operations in hardware, but its critical path is not short because, for instance, the number of cascaded 8x8 lookup tables is 48, where that for Rijndael is 10 when the key length is 128 [B.Schneier et. al. (1998)].

3. Design Policy

Our purpose is to evaluate the fastest possible encryption speed of the AES finalists using the existing hardware library under fair conditions. To achieve this and also to complete the analysis in our limited time scale and resources, we designed the 128-bit key version for each candidate on the basis of the following criteria and conditions:

1. We fully unrolled the loop in the encryption and decryption logic and the key scheduling logic to achieve the fastest possible speed (throughput). In practice, the loop structure is commonly used in order to reduce hardware size, but generally makes the hardware slower because additional setup -time and hold-time is required for the loop registers, which is usually not negligible. Note that we therefore did not take a special effort to reduce hardware size.

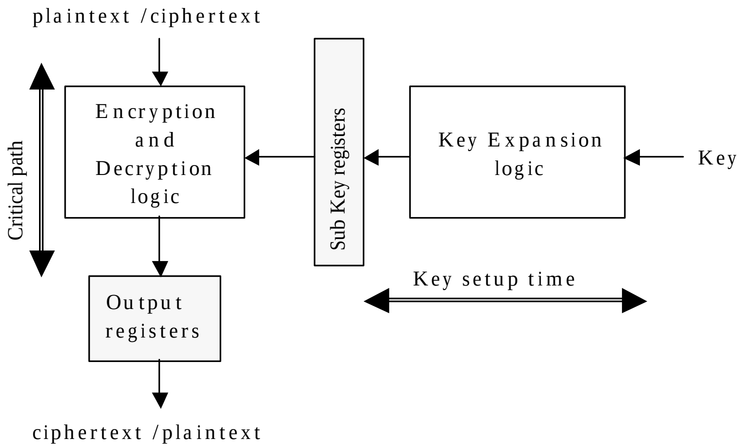


Figure 3.1 The hardware structure

2. We assume that all subkey bits are stored in subkey registers before an encryption operation begins. Also we have inserted another 128-bit resister to hold a block of ciphertext as shown in Figure 3.1, where we define the critical encryption and decryption path as the time required for all output bits of the encryption and decryption logic to reach the output registers under the fixed (sub)key value.

3. We did not introduce pipeline architecture; i.e., we did not insert any additional intermediate registers in the encryption and decryption logic. This is because the pipeline architecture makes the ECB mode faster but also blocks feedback modes of operations such as CBC. In other words, our hardware model encrypts one block plaintext data in one cycle.

4. We did not use a special optimization technique to design lookup tables in hardware. This means that the performance of the lookup tables heavily depends on optimization capability of the logic synthesis tool. In practice, as will be shown in the next section, the output of the synthesis tool seems to have reasonably optimized the lookup tables (not very slow).

5. Our design environment is as follows:

language:	Verilog-HDL
simulator:	Verilog-XL
design library:	Mitsubishi 0.35micron CMOS ASIC Library
logic synthesis:	Synopsys Design Compiler version 1998.08

For arithmetic operations such as additions, subtractions and multiplications, we used faster ones in the library of Synopsys Design Ware Basic Library [Synopsys (1998)]. Also, we adopted the WORST case hardware conditions for evaluation. The worst case speed is a guaranteed speed of a given circuit, which is commonly used in real products. We think that the TYPICAL case evaluation is too optimistic to apply to a real ASIC hardware.

4. Evaluation Results

The results of our hardware evaluation of the five finalists are presented in Table 4.1. The fastest algorithm in terms of the critical path between plaintext and ciphertext is Rijndael, which is an only algorithm faster than DES. The second fastest algorithm is Serpent, which is twice faster than triple-DES but still much slower than Rijndael (approximately half). The speed of Twofish is almost the same as that of triple-DES, but Mars and RC6 are further slower; Rijndael is approximately ten times faster than RC6.

On the other hand, for the key setup time, Twofish is fastest, consuming only 5% of the critical path of its encryption procedure. Note however that the key setup time of DES and Triple-DES is almost nothing in hardware. Rijndael and Serpent have approximately 85%, while the key scheduling logic of Mars and RC6 is more than three times slower than their encryption.

Figures 4.1 and 4.2 show more detailed breakdowns of hardware components on the critical path of each algorithm, where the horizontal line of Figure 4.2 is normalized to show proportion of each component .

Mars has 16 multiplications, 26 additions/ subtractions, 15 lookup tables (specifically 11 S0's and 4 S1's) and 9 data-dependent rotations on its critical path, where all arithmetic operations are taken on mod 2^{32}. As shown in the figures, the multiplications occupy 63% of the critical path, 13% for additions/subtractions, and 9% for the lookup tables.

RC6 has 20 multiplications, 21 additions and 20 data-dependent rotations on its critical path, where all arithmetic operations are also taken on mod 2^{32} As shown in the figures, the multiplications occupy 77% of the critical path, 13% for additions/subtractions, and 8% for the data dependent rotations.

The critical path of Rijndael is not in the encryption but in the decryption procedure since the InvMixColumn function, which is an inverse of the MixColumn function, is a bit slower than the MixColumn function due to more complex constant values. On the critical path, a total of 10 InvByteSub functions (table lookups) occupy 48% of the entire decryption time, and a total of 9 InvMixColumn functions have 43%.

It is easy to see that the critical path of Serpent has 32 lookup tables and 31 linear transformations (XOR's and shifts). Our analysis shows that the linear transformations of Serpent are more expensive than its lookup tables; the former is 36% while the latter is 45%. In a logical sense, the lookup tables and the linear transformations must exhaust the critical path; however Figure 4.2 exhibits other factors that occupy a total of 19%. This is mainly because the design compiler has automatically inserted driver gates in order to supply sufficient fan-out counts, which reflects the fact that an output bit of a lookup table of Serpent has many "branches" that reach many different lookup tables in the next round. This is part of design criteria of Serpent.

It is also easily seen that the critical path of Twofish have 48 lookup tables --- specifically 16 q0's and 32 q1's, which is not a trivial fact ---, 16 MDS's (linear transformations) and 32 additions mod 2^{32}. The dominant part is the lookup tables, which occupy 53%, but also time for additions is not negligible (28%).

5. Discussions and Conclusions

The performance of Mars and RC6 heavily depends on the speed of the multiplication circuits mod 2^{32}. Our evaluation results show that the average time for the multiplication is around 23ns, which is six to eight times slower than the addition circuit mod 2^{32}, which takes around 3ns.

This also shows that by using highly optimized multiplication circuits in a transistor level, these algorithms are expected to be much faster. For this topic, see [Hagi (1998)] for instance. Now as an example, let us assume, in Mars and RC6, the 32-bit multiplication can work at the same speed as the 32-bit addition. We see that still the critical path of (the modified) Mars and RC6 is approximately 250 and 200ns, respectively. Also, we should notice that a full-custom solution is generally process-dependent and hence is not an inexpensive solution in practice.

Another speeding-up possibility is to optimize a lookup table. The average time for one lookup table for each algorithm is 3.2ns for Rijndael (8x8), 1.5ns for Serpent (4x4), 3.5ns for Twofish (8x8) and 3.5ns for Mars (8x32), respectively. Twofish will be most rewarded for the efforts of optimizing the lookup tables. However, the optimization will not lead to a significant impact to affect the ranking of the five finalists.

In this paper, we did not take efforts to reduce the size (area) of each algorithm since we adopted a full loop unrolling in order to evaluate the fastest possible encryption speed. Appendices 1 and 2 show the information of the size of each algorithm with the detailed breakdowns, which we will not discuss here. How to reduce the gate size is another practical topic to be pursued.

References

[NIST (1998)]:
http://csrc.nist.gov/encryption/aes/aes_home.htm

[MITSUBISHI (1997)]:
Mitsubishi Electric America, Inc.,
"0.35um CMOS ASIC DATA BOOK ", 1997.

[Burwick et.al. (1999)]:
http://www.research.ibm.com/security/Mars.html
See also
http://csrc.nist.gov/encryption/aes/round2/AESAlgs/MARS/mars-int.pdf

[Rivest (1998)]:
R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 Block Cipher," 1998.

[RSA(1998)]:
http://www.rsasecurity.com/rsalabs/aes/index.html

[J.Daemen and V.Rijmen (1998)]:
J. Daemen and V. Rijmen, "AES Proposal: Rijndael," Document vers on 2, Date: 03/09/99.
http://www.esat.kuleuven.ac.be/~rijmen/rijndael

[Biham (1997)]:
E Biham, "A Fast New DES Implementation in Software", in Fast Software Encryption - 4th International Workshop, FSE '97, Springer LNCS v 1267,pp 260-271.

[Anderson, Biham and Knudsen (1998)]:
R. Anderson, E. Biham and L. R. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," 1998.
http://www.cl.cam.ac.uk/~rja14/serpent.html

[B.Schneier et. al. (1998)]:
B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Fergusen, "Twofish: A 128-Bit Block Cipher," June 15, 1998.
http://www.counterpane.com/twofish.ps.zip

[Synopsys (1998)]:
Synopsys Inc. ,"Design Ware Foundation Quick Reference Guide ", Aug.1998.

[Hagi (1998)]:
Y.Hagihara, et. al., "A 2.7ns 0.25um CMOS 54x54b Multiplier", ISSCC Digest of Tech. Papers, pp296-297, Feb.1998.

Table4.1 Hardware evaluation results

Algorithm name	area [Gate]			Key setup time[ns]	Critical-path[ns]	Throughput [Mbps]
	Encryption & Decryption	Key Schedule	Total			
DES	42,204	12,201	54,405	-	55.11	1161.31
Triple-DES	124,888	23,207	148,147	-	157.09	407.4
MARS	690,654	2,245,096	2,935,754	1740.99	567.49	225.55
RC6	741,641	901,382	1,643,037	2112.26	627.57	203.96
Rijndael	518,508	93,708	612,834	57.39	65.64	1950.03
Serpent	298,533	205,096	503,770	114.07	137.4	931.58
Twofish	200,165	231,682	431,857	16.38	324.8	394.08

Figure 4.1 Critical Path of the Finalists(1)

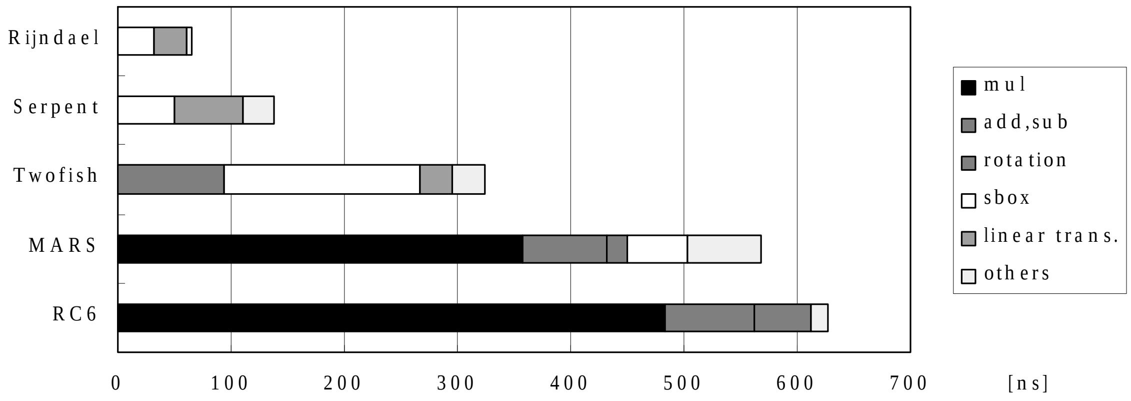
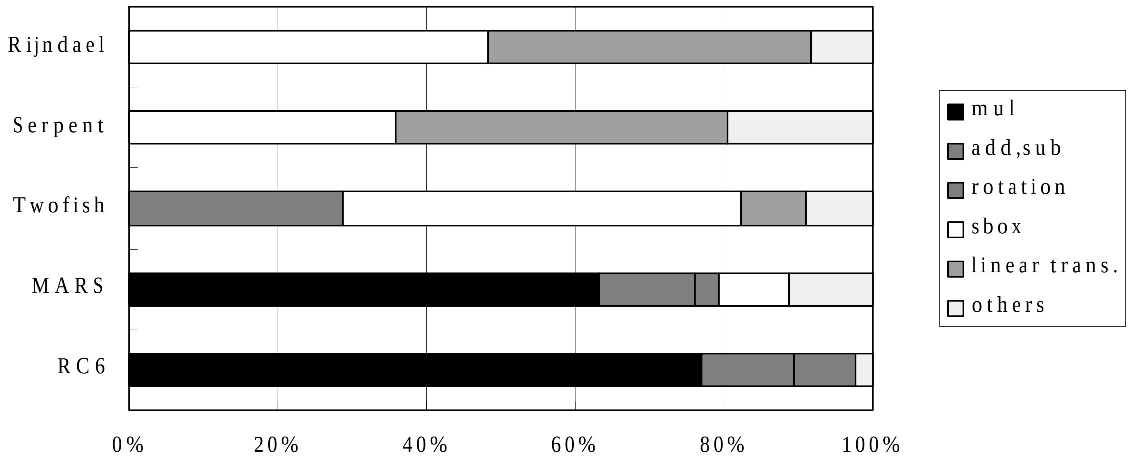
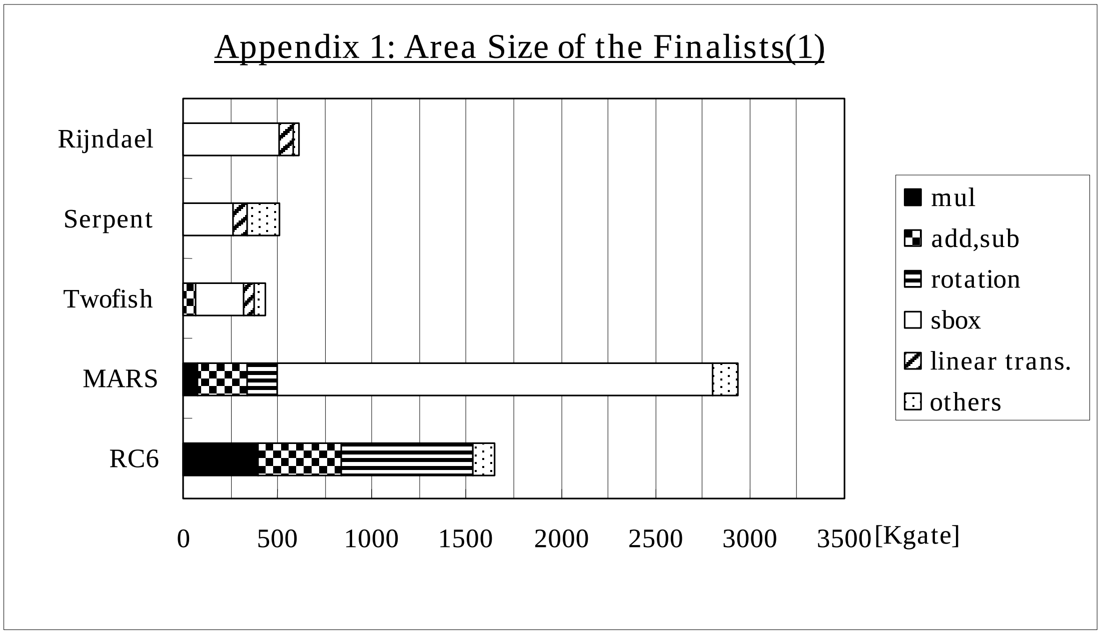
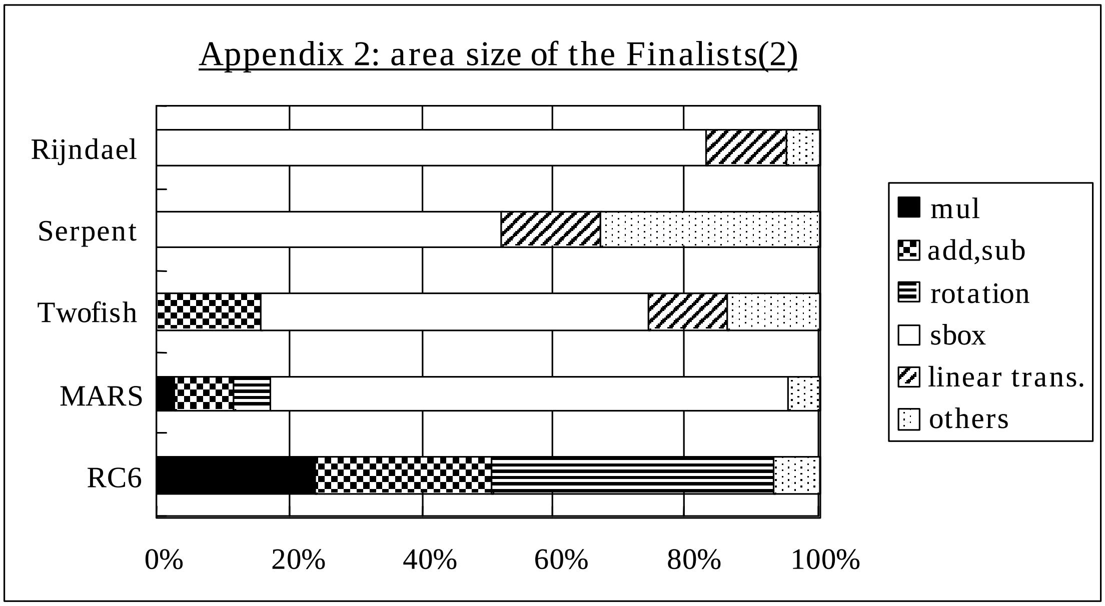


Figure 4.2 Critical Path of the Finalists(2)




Appendix 1: Area Size of the Finalists(1)


Appendix 2: area size of the Finalists(2)

Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms

Bryan Weeks, Mark Bean, Tom Rozylowicz, Chris Ficke

National Security Agency

1 Abstract

The National Security Agency (NSA) is providing hardware simulation support and performance measurements to aid NIST in their selection of the AES algorithm. Although much of the Round 1 analysis focused on software, much more attention will be directed towards hardware implementation issues in the Round 2 analysis. As NIST has stated, a common set of assumptions will be essential in comparing the hardware efficiency of the finalists. This paper presents a technical overview of the methods and approaches used to analyze the Round 2 candidate algorithms (MARS, RC6, RIJNDAEL, SERPENT and TWOFISH) in CMOS-based hardware. Both design procedures and architectures will be presented to provide an overview of each of the algorithms and the methods used. To cover a wide range of potential hardware applications, two distinct architectures will be targeted for comparison, specifically a medium speed, small area iterated version and a high speed, large area pipelined version. The standard design approach will consist of creating hardware models using VHDL and an underlying library of cryptographic components to completely describe each algorithm. Once generated, the model can be verified for correctness through simulation and comparison to test vectors, and synthesized to a common CMOS hardware library for performance analysis. Hardware performance data will be collected for a variety of design constraints for each of the algorithms to ensure a wide range of measured data. A summary report of the findings will be presented to demonstrate algorithm performance across a wide range of metrics, such as speed, area, and throughput. This report will provide a common baseline of information, which will enable NIST and the community to compare the hardware performance of the algorithms relative to one another.

2 Introduction

The National Security Agency (NSA) agreed to provide technical support to the National Institute of Standards and Technology (NIST) in the form of an analysis of the hardware performance of the Round 2 Advanced Encryption Standard (AES) algorithm submissions. This analysis consisted of the design, coding, simulation and synthesis of the five algorithms using the procedure outlined below. Throughout this evaluation, NSA has taken care to assure that best design practices were used and that all algorithms received equal treatment. No attempt was made to optimize any particular design, but care was taken to find the best configuration for each algorithm. Cross-validation measures during design and simulation were used to overcome the subjective effects of the design process and to ensure that all designs receive the same amount of attention. The results of this analysis should provide an accurate measure of the hardware performance of each algorithm relative to the others. Undoubtedly more optimized (and hence better performing) implementations of these algorithms can be designed, so the individual score of any particular algorithm is not very valuable outside the context of this environment. The point of this analysis is to provide a controlled setting in which a meaningful comparison can be made.

Based on a mathematical description of the Round 2 algorithms, and C code reference models when necessary for clarification, NSA designers fully described each of the algorithm submissions in a hardware modeling language. A review by a team of design engineers followed the initial design stage to reduce the effects of coding style on performance. Using commercially available analysis, simulation and synthesis tools, NSA design engineers have performed simulations to produce performance estimates based on each of the hardware models. In order to provide a wider perspective on the performance of the algorithms, two different architectures or applications were simulated for each algorithm: an iterative version to provide a medium speed operation at minimal area/transistor count, and a pipelined version to provide optimum speed operation, but at the cost of a larger area. This report is a summary of the performance of the Round 2 AES candidate algorithms, and will compare and contrast the results of the analysis.

3 Hardware Design Background

3.1 Design Guidelines

For this analysis effort, one of the main goals was to provide an unbiased comparison of the algorithms in hardware, specifically in Application Specific Integrated Circuits (ASICs). To that end, the overhead found in typical hardware implementations, such as a robust user-interface, was minimized to reduce the impact on the overall performance of the algorithms. The user-interface is the Input/Output (I/O) connections and logic needed to take the plaintext and key and present them to the algorithm, and take the output ciphertext and present it off the chip. All inputs and control signals were registered in a common interface in order to provide uniformity across all of the algorithms, with fixed setup and hold times identical for all algorithms. A wide variety of architectures could be used to implement a given algorithm. In order to restrict all possible choices and yet capture valuable data points, two fundamental architectures were chosen: iterative and pipelined. All algorithms were designed in each architecture style. There are several variations on these approaches, including multiple copies of an iterative implementation for parallel processing, a partially pipelined implementation, or a combination of these hybrids (multiple copies of a partially pipelined implementation). The approach chosen will depend on the needs of the system, but these variations will likely result in performance within the ranges given by the iterative and fully pipelined implementations. However, these optimizations were beyond the scope of this study.

3.1.1 Target Applications

3.1.2 Iterative Architecture

The iterated approach to implementing the algorithm focuses on providing a medium to low speed version of the algorithm, with efforts placed on limiting the physical size of the hardware. In this instance of the algorithm, one step is performed per clock period, with the output of the previous step being used as the input to the next step. Data is only placed on the output after the required number of algorithm rounds has been completed.

3.1.3 Pipelined Architecture

The pipelined approach to implementing an algorithm centers on providing the highest throughput to the design, sacrificing area to obtain the level of performance needed. In the case of pipelining, all of the steps in computing the algorithm are cascaded into a single design, with each stage feeding the next stage. The latency remains the same as in the iterated case, but the throughput is increased significantly as new data is placed on the output on every clock cycle. Pipelining has been shown to be an effective method of dramatically increasing the throughput capabilities of a given algorithm. However, it comes at the expense of limiting the number of cryptographic modes that can be supported at the maximum throughput rate. For example, since the latency of an encryption cycle remains the same as an iterative case, there is no throughput advantage when using feedback modes such as Cipher Block Chaining (CBC). High performance applications, such as high speed network encryption, will require the increase in throughput, and as a result, often focus on a non-feedback mode of operation such as counter mode to obtain performance.[1]

3.2 Parameter Description

There are many design parameters that can be reported for each design implementation. Some parameters will have much more significance in a given application or environment than others. This evaluation reports on these parameters as a method of comparison among the five algorithms, and does not claim that any single parameter has been fully optimized. The following is a description of the parameters being reported. Some have a direct impact or relation to performance metrics (e.g. throughput) and some are simply a function of the algorithm itself (e.g., I/O requirements). Algorithm performance in each of the evaluation categories will be documented for each algorithm submission.

3.2.1 Area

As an estimate based on an available MOSIS library, the results of the synthesis area reporting will consist of pre-layout area estimates of the algorithm. Although potentially different from a post-layout estimate, the area reported by Synopsys will provide a relative comparison of each of the algorithm submissions. Generally, the two varieties of architectures-– iterative and pipeline -– will be on the extremes of area with the iterative being the smallest, and the pipelined being the largest.

3.2.2 Throughput

In most cases, throughput is directly proportional to area; as area decreases, throughput decreases. As with area, the iterative and pipelined architectures will report the extremes of throughput. Iterative architectures will have much lower throughput rates since there is a minimum amount of hardware, and it is re-used on multiple clock cycles of execution. Thus, the throughput is limited by the amount of hardware reuse. More specifically, it is limited by the number of rounds in a codebook algorithm. On the other hand, a pipelined architecture dedicates hardware for performing all calculations in any given clock cycle. This maximizes throughput by allowing data to be written and read from the device on every clock. In this case, throughput is a function of the worst-case delay in any one given stage of the algorithm. Throughput will be reported for both iterative and pipelined architectures.

3.2.3 Transistor Count

Transistor count is a more specific measure than area and is often more useful. While transistor count is somewhat dependent on the design library being used, it is a useful method of comparing the algorithms since they were compiled using the same library. In addition, the transistor count will be a more useful figure than area when estimating programmable logic implementations since these devices typically report the number of useable gates (which is also directly related to transistor count). Based on the synthesized netlist (from Synopsys), an additional report describing the number of transistors required to implement the algorithm will be provided.

3.2.4 Input/Outputs (I/O) Required

With the goal of consistency among algorithms, the I/O was fixed identically for all algorithms. However, since this parameter is highly useful to hardware designs, it will still be reported.

3.2.5 Key Setup Time

The key setup time refers to the amount of time required before subkey expansion is ready to execute. Some algorithms use the user-supplied key directly in the subkey expansion thereby reducing the key setup time to zero. Others require some pre-calculation or translation of the key prior to subkey expansion steps. Key setup times will be examined to assess the overhead of each algorithm in establishing a usable key.

3.2.6 Algorithm Setup Time

Similar to key setup time, the algorithm setup time reports the minimum amount of time before an algorithm is ready to process data. Time to create look-up tables, etc. will fall in this category. None of the evaluated algorithms contained an algorithm setup time greater than zero.

3.2.7 Time to Encrypt One Block

This paramter will address minimum latency times for each of the algorithm submissions. The time to encrypt one block, measured in nanoseconds, is a function of two parameters: the worst-case path delay between any two registers, and the number of rounds in the algorithm.

3.2.8 Time to Decrypt One Block

As above, this parameter will address minimum latency times for each of the algorithm submissions. Decryption does not always require identical processing as encryption. Therefore, the time required to decrypt one block is reported.

3.2.9 Time to Switch Keys

Originally, this parameter was included as a measure to encompass both key setup time and algorithm setup time overhead. However, since none of the evaluated algorithms contained an algorithm setup time, this parameter is identical to key setup time. Therefore, it will not be reported further in this document.

4 Methodology

4.1 Standard Design Flow

The design process followed a common methodology used by ASIC designers. The process started with the documentation supplied by the algorithm authors and was completed with a gate-level schematic, which included the performance metrics data. A complete ASIC development would require physical layout and fabrication. These steps were beyond the scope of this effort. However, the performance metrics data obtained here closely matches that which would be found from actual fabrication and testing. Previous efforts using these tools have correlated estimated performance from the schematic to the actual testing. Figure 1 shows the steps in the design flow.

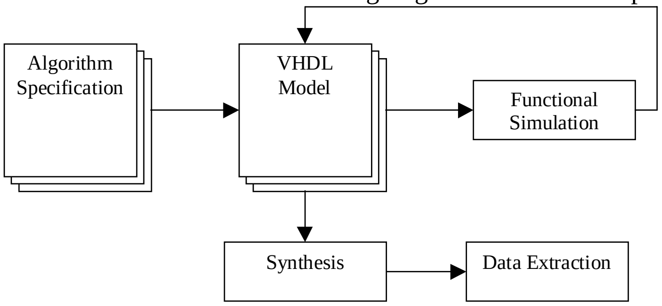


Figure 1 Standard Design Flow

4.1.1 VHDL code generation

VHSIC Hardware Description Language (VHDL)

VHDL modeling is analogous to programming simulations in C code and follows much of the same syntax. However, unlike a behavioral description of the algorithm, VHDL (IEEE 1076) specifies how the algorithm will be implemented in hardware. Using this hardware language, NSA designers fully described the hardware necessary to implement each of the algorithm submissions. Performance metrics, such as speed, area, etc. (see below) can be estimated from the hardware description using available analysis and computer aided design (CAD) tools.

There are different styles in which to code VHDL models, offering various levels of abstraction. For this evaluation, the designers used the register transfer logic (RTL) coding style. For this style, the placement of registers and corresponding logic between registers is chosen by the designer and is determined at the VHDL code level. There are many different methods for identifying an optimized placement of registers. Ideally, there would be an equal amount of logic delay between registers for all stages of the design. However, in order to simplify the design cycle and to be consistent among all algorithms, the designers chose a common placement of registers, even if this placement is not fully optimized. Specifically, the output of each “round” (as defined by the algorithm authors) is registered for both a key schedule round and an algorithm round.

4.1.2 Simulation and Verification

NSA followed the design phase with a functional VHDL simulation of the designs using the Synopsys VHDL System Simulator (VSS) to verify the correct operation of the algorithm. The test vectors submitted to NIST for each algorithm were applied to assure that the design was working as intended. Specifically, the Variable Key and Variable Text tests were performed for each algorithm implementation and mode (e.g., iterative encrypt, pipelined decrypt, etc.). The modeled algorithm output was also compared with the C code model supplied to provide an added assurance that the simulation was operating as expected.

4.1.3 Code review

NSA had one or more engineers design the VHDL for each algorithm submitted. Initial hardware designs were straightforward implementations of the core algorithm. Following completion of each initial design, an informal group of engineers met to review and provide feedback for the design. Improvements and alternatives to the initial design were examined to determine potential benefits from differing architecture approaches (area compression, pipelining, etc.). Variants of the design that improve the performance of the algorithm were then programmed for comparison.

4.1.4 Synthesis

Gate-level synthesis of the algorithm utilized the Synopsys Design Compiler to produce a functionally equivalent schematic in hardware. A MOSIS-specific technology library was used to generate a gate-level schematic of the design and provide more accurate area and timing estimates, as if the design were to be implemented in an integrated circuit (IC). The MOSIS library is based on a publicly available fabrication facility's model of a specific CMOS process, thus giving real performance metrics for an available ASIC line. The VHDL model can be re-targeted to any supported hardware or field programmable gate array (FPGA) design libraries.

The synthesis process can generate a wide range of implementations depending on the constraints provided to the synthesis tool. For example, one implementation may minimize area while another may minimize delay time. In hardware synthesis, the two fundamental parameters are time and area. These parameters are directly related. As delay time decreases, area increases. Timing and area curves that further illustrate this point are shown in subsequent sections. The constraints provided for each algorithm synthesis routine were maintained consistently. Therefore, differences among algorithm synthesis results will be a function of the logic required (algorithm specific) and the synthesis tool's ability to meet the given constraints.

4.1.5 Documentation

In addition to a summary report containing performance data, both design notebooks and VHDL documentation will be provided to NIST for evaluation. The design notebooks will contain reporting information for all of the hardware data that was collected, with all algorithms, designs, and architectures represented. The VHDL models and their testbenches (for simulation verification) will also be included.

4.2 Synthesis Analysis

4.2.1 Function Characterization

Although the hardware design of the algorithms followed a top-down approach, the synthesis portion of the analysis proceeded from the bottom of the design up through the hierarchy. In order to obtain an accurate picture of the performance of the sub-blocks and functions in this type of analysis, a sweep was performed on each of the functional blocks to graphically depict performance versus design constraints. Specifically, the timing constraints, such as output delay and clock frequency, of each of the blocks were varied to observe the performance output of the block. The results of the sweep make up the characterization for that particular block. All subsequent blocks of the hierarchy will be analyzed using these methods.

4.2.2 Cryptographic Library

With characterization curves for each of the sub-blocks complete, five speed grade implementations were selected to cover the performance range of the block. A variety of key performance points were selected to reflect requirements for both high speed and small area. Figure 2 shows typical timing and area curves following a sweep of maximum delay time constraints. These curves allow design engineers to select specific implementations of a given function. Specifically, five implementations, or speed grades, were chosen for each function. In this example, the five selected implementations are noted in the figure, and they represent one minimum time delay, one minimum area, and three other points that have desired characteristics such as large area savings for a small increase in delay time.

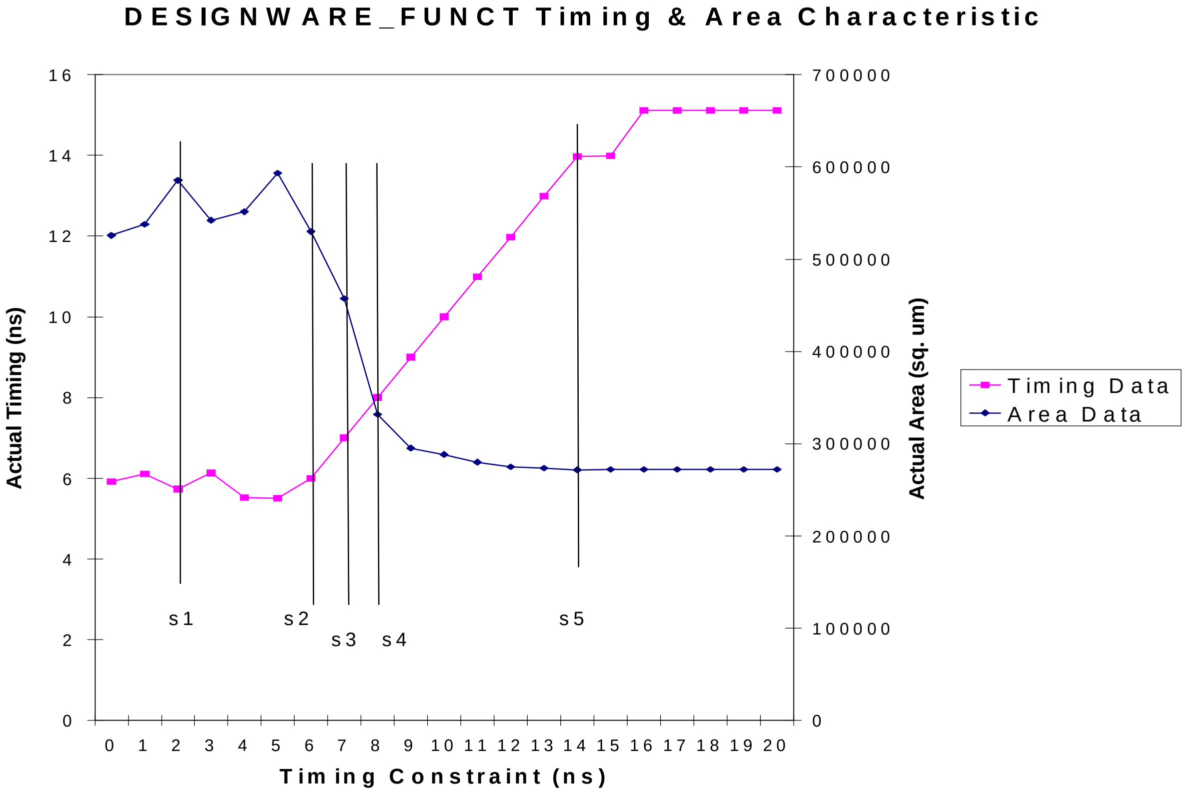


Figure 2 Sample Function Sweep

The five implementations were selected only for the functions of each of the algorithms, and then assembled into a cryptographic library. Each library contained implementations of all the functions required to build the given algorithms.

4.2.3 Block Level Characterization

Continuing with the bottom up approach, the higher level blocks underwent the same performance sweep as described for the function level. The design constraints were varied across the entire range of the block to fully describe the performance curve of the block. At each iteration of the synthesis process, components (e.g., functions) were selected from the cryptographic library based on the required speed and performance. Performance curves at the block level encompassed components of several different speed grades depending on design constraints. At the top level, the characterization curve reflected the performance of the entire design across a wide range of design constraints.

5 General Architecture Approach

5.1 Top Level Architecture

The design of each algorithm started with a common top level architecture that is well suited for virtually any codebook algorithm. The generalized top level architecture consists of an Interface, an Algorithm block, a Key Schedule block and a Controller. Figure 3 shows a block diagram of the architecture.

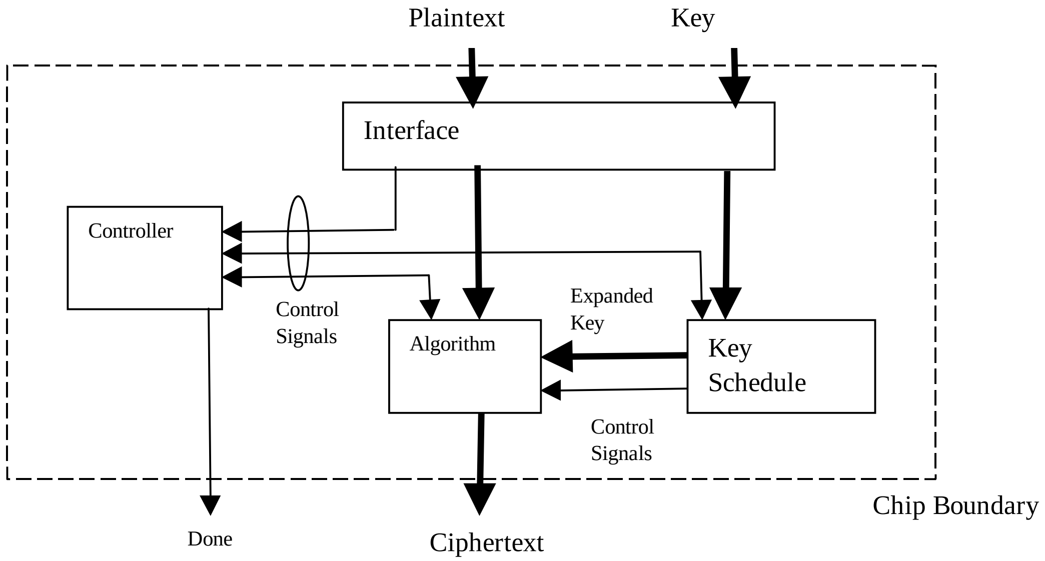


Figure 3 Top Level Architecture

The Interface serves to register all data inputs. This is consistent with the hardware design methodology of placing registers at the chip boundary, thus minimizing strict setup-and-hold timing requirements. In some cases, the interface also provides minimum functionality, such as padding keys when appropriate. The Key Schedule performs the generation of subkeys to be used in by the Algorithm block. This includes any required key setup as well as the expansion itself. The Algorithm performs the actual encryption or decryption of data provided from the Interface using the subkeys from the Key Schedule. For iterative implementations, the Algorithm and Key Schedule blocks implement a single round with internal feedback datapaths; whereas the pipelined implementations expand these sections to include as many implementations of a round as required by the particular algorithm. Finally, the Controller provides any necessary control signals for maintaining proper synchronization among the various blocks.

6 Algorithm Evaluation

For each of the algorithms, a description of how it was architected for both the pipelined and iterated cases is given. Any nuances of how the rounds were simulated and the key schedule implemented are also given along with specific examples of approaches to reduce redundancy or streamline the design. Each algorithm section then provides block level results of timing constraints versus both chip area and timing in both the iterative and pipelined cases. A table of performance parameters is then provided for four different key sizes, 128 bit key, 192 bit key, 256 bit key, and a hybrid that combines all three key sizes in one key schedule that can be controlled for any particular key size. In some cases, the combined three-in-one key schedule must make compromises to achieve the greater degree of flexibility. Each of the performance parameters is described in more detail in Section 7, along with comparisons across the five algorithms.

Following the architecture for each of the algorithms, each section will provide a summary of the results of the hardware analysis for the individual algorithm. In an effort to save space, the timing and area graphs will be presented for only the combined case which contains all three key sizes in one implementation. Both pipelined and iterated cases will be covered. However, the complete report and design workbooks will contain graphs for all key sizes and contains a much more complete data set. The corresponding tables will capture key performance data points for all key size implementations. *

* Note: At time of publication, not all information was available for every parameter and for every algorithm. Due to some unforeseen difficulties in the amount of time for simulation, some information on area, transistor count and key setup times was not available. This was especially true for simulating the larger blocks in the pipelined cases and for the various key sizes. In addition, certain information for MARS and RC6 was being finalized at time of publication, so is not included in this version. Incomplete data in the following sections are indicated with asterisks.

Complete data for the performance curves and tables of key parameters will be provided on the NIST web site and at the conference.

6.1 MARS

6.1.1 Architecture

The MARS algorithm requires several different types of rounds[2]. Specifically, there are unkeyed forward mixing, keyed forward transformation, keyed backwards transformation and unkeyed backwards mixing rounds, as well as pre-addition and post-subtraction. The mixture of keyed and unkeyed rounds resulted in the requirement for complex control and data flow operations between the Key Schedule and Algorithm blocks. Specifically, a complex control situation results from the fact that subkeys are required immediately for the pre-addition stage, whereas the next subkeys are not required until the eight unkeyed forward mixing rounds are completed. This architecture presented some unique timing and data synchronization issues.

6.1.1.1 Pipelined Key Schedule

As with all pipelined implementations, the subkey from each round must be registered. However, for MARS, the subkeys are not utilized on consecutive clock cycles. Therefore, additional pipelined storage is necessary. The updated key schedule of MARS following AES Round 1 allowed for separating the 40 subkeys into groups of 10. The VHDL model takes advantage of this operation by adding pipelined storage for groups of 10 subkeys, only as long as necessary, rather than creating pipelined storage for all 40 subkeys. This reduced the total number of registers required. Additionally, the pipelined registers are controlled by a latch signal rather than updating on every clock. Again, this reduced the number of registers by removing redundancy.

6.1.1.2 Pipelined Algorithm

Relative to the intricacies of the key schedule, the pipelined algorithm implementation is straightforward. It consists of six different types of rounds, each one with its own registered output: one key addition, eight unkeyed forward mixings, eight keyed forward transformations, eight keyed backwards transformations, eight unkeyed backwards mixings and one key subtraction. This makes a total of 34 rounds to complete the algorithm.

6.1.1.3 Iterative Key Schedule

The MARS algorithm key schedule generates 10 subkeys at a time. Therefore, the traditional iterative methodology of a single round implementation for generating a single subkey (or set of subkeys as required by a single algorithm round) did not apply. Instead, a single round implementation per 10 subkeys was generated resulting in a key expansion round iterated four times for one encryption cycle. This presents some additional logic overhead for iterative applications in that a “round” generates 10 subkeys simultaneously rather than the exact amount needed by the algorithm at a given stage. (In the case of MARS, two 32-bit subkeys are required per keyed round in the cryptographic core.) In addition to the subkey expansion overhead, there is a storage overhead for the remaining subkeys. Also, decryption requires a full expansion of subkeys prior to beginning data processing. Therefore, the full set of 40 subkeys is stored in registers.

6.1.1.4 Iterative Algorithm

The iterative algorithm is consistent with the pipelined algorithm in its relative simplicity when compared to the key schedule. There is a single register for all rounds. The input to the register depends on the round number. For example, the input for the first round of encryption is the key addition round result; for the second round it is the unkeyed forward mixing round result and so on.

Subkeys are presented to the algorithm block as an array of all 40 subkeys. This differs from other iterative algorithm implementations that present only one subkey at a time. The rationale for this design was to eliminate duplicate logic in both the Key Schedule block and Algorithm block. Due to the timing gaps in the application of subkeys and the fact that all 40 subkeys are generated prior to decryption processing, it was considered advantageous to allow a 40 element bus to connect the two blocks.

6.1.2 MARS Top Level Results

6.1.2.1 Timing and Area

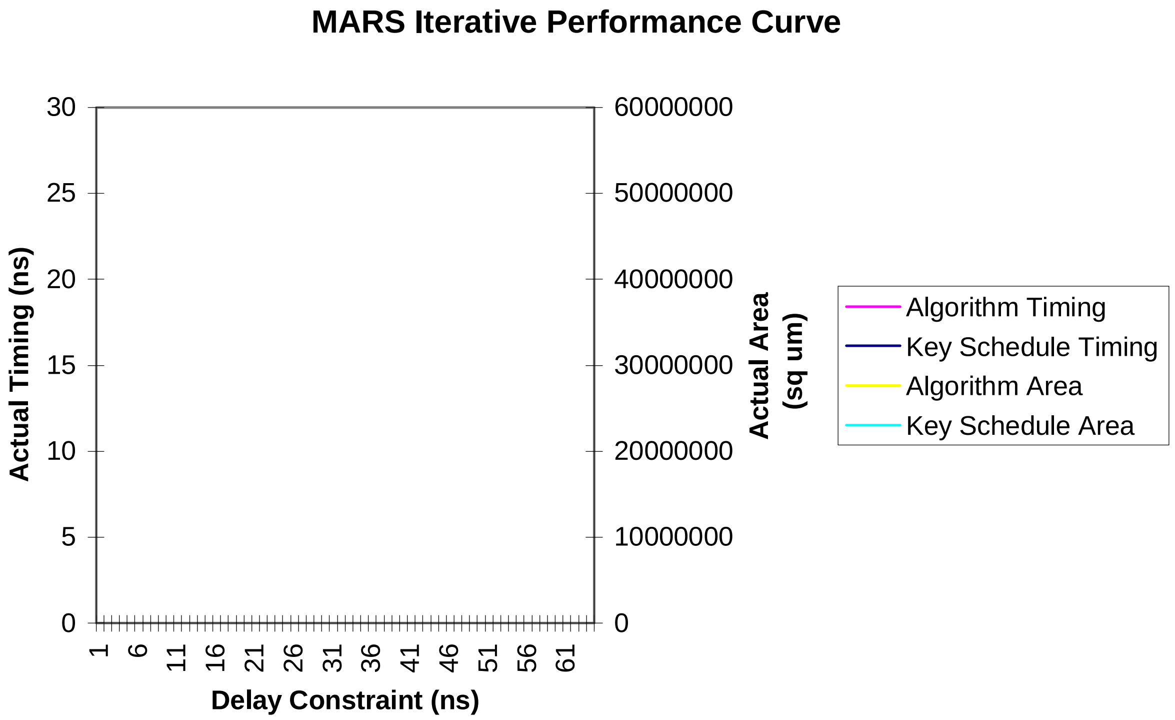


Figure 4

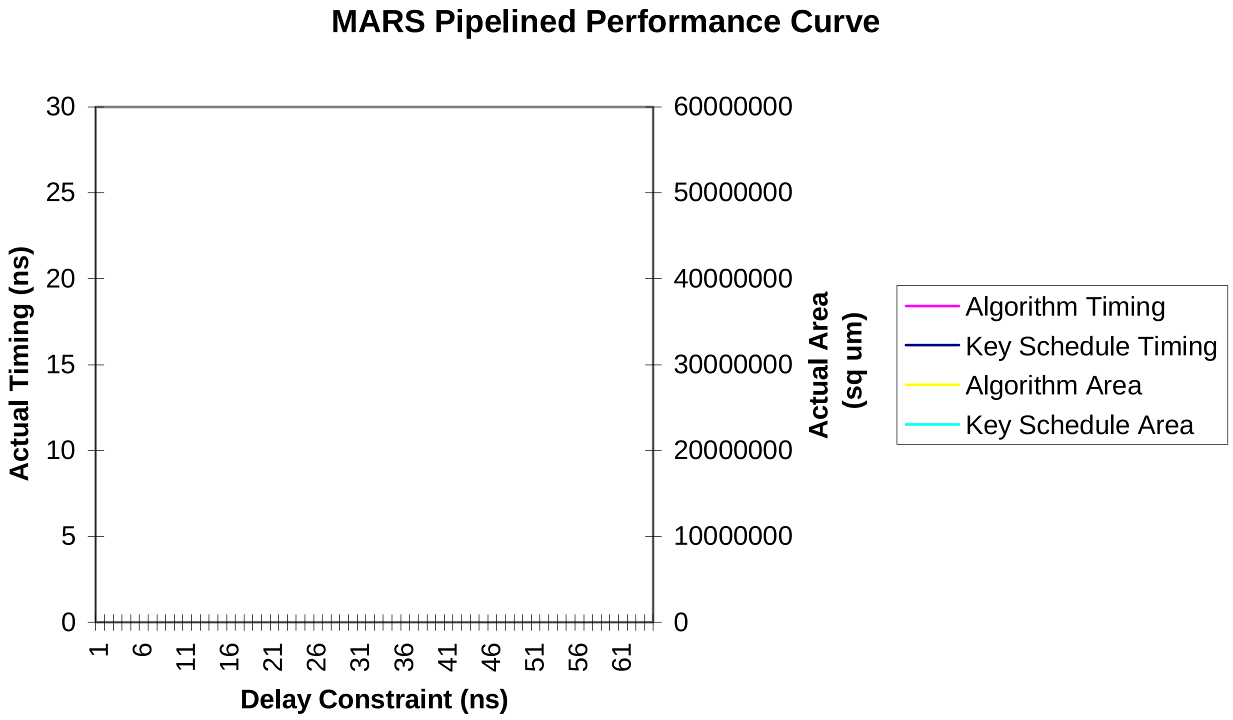


Figure 5

6.1.2.2 Key Parameters

Parameter	Iterative 3in1		Pipelined 3in1	
	Min.	Max.	Min.	Max.
Area (um2)	*	*	*	*
Transistor Count	*	*	*	*
Input/Outputs Required	520	520	520	520
Throughput (Mbps)	*	*	*	*
Key Setup Time Encrypt (ns)	*	*	*	*
Key Setup Time Decrypt (ns)	*	*	*	*
Algorithm Setup Time (ns)	0	0	0	0
Time to Encrypt One Block (ns)	*	*	*	*
Time to Decrypt One Block(ns)	*	*	*	*

Table 1 MARS Summary

6.2 RC6

6.2.1 Architecture

The following provides a high level description of the major blocks in the RC6 algorithm. Details of the components, sweeps, and their implementations can be found in the design workbook[3].

6.2.1.1 Pipelined Key Schedule

The RC6 key schedule is pipelined using a slightly different method than the other algorithms. Since a significant number of computations for the key schedule are required before any expanded keys are generated, the architecture takes advantage of the run-up by performing the expansion at the start of the pipeline. Only a single copy of the expansion hardware is required, but additional registering is needed to maintain the keys on a time dependent basis, discarding keys from previous stages (i.e., the keys have already been used). Keys are then passed from register to register to follow the data in the pipeline.

6.2.1.2 Pipelined Algorithm

The algorithm “unrolls” the stages of the algorithm into a pipeline, following the algorithm description for function ordering and naming conventions. Combination functions are used to perform cases where distinct operations need to be performed in encrypt and decrypt. For example, the pre-add will contain both addition and subtraction to accommodate both cases. A similar condition exists in the algorithm round function, with slightly different functions needed for encrypt and decrypt. However, synthesis optimization can take advantage of common operations, such as the multiply, to reduce the total number of operators needed.

6.2.1.3 Iterative Key Schedule

The iterative key schedule is designed to perform a single round of expansion per clock. Expanded keys are fed to the algorithm block after the controller initiates a start signal. However, the key setup has been designed to compute single or multiple steps of the run-up in a single clock, depending on the performance needed by the rest of the system. A load cryptovariable (i.e., load key) signal from the controller will initiate the key setup. Once complete, the expansion can be started.

6.2.1.4 Iterative Algorithm

The RC6 iterative algorithm closely reflects the pipelined version. The same round instance is called repeatedly to process input data. The encrypt and decrypt are symmetrical with respect to operations performed in a similar manner (e.g., pre-add, round, post-add), so the same block can be called without additional overhead.

6.2.2 RC6 Top Level Results

6.2.2.1 Timing and Area

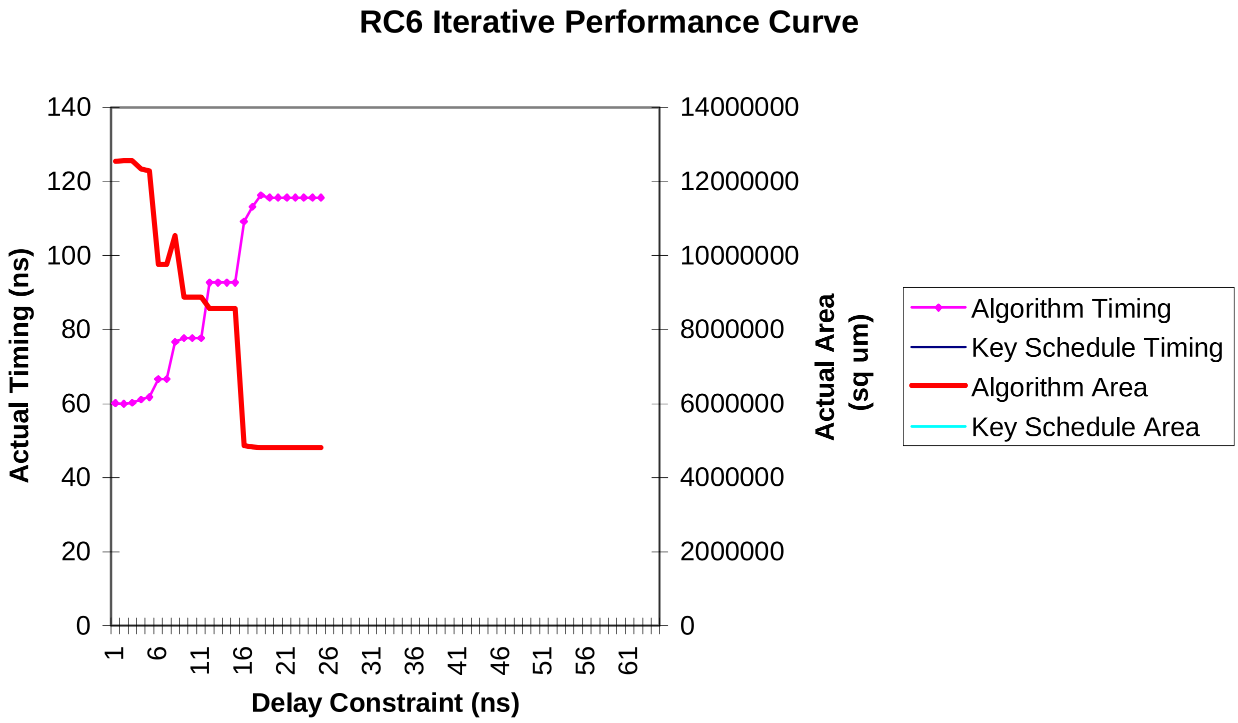


Figure 6

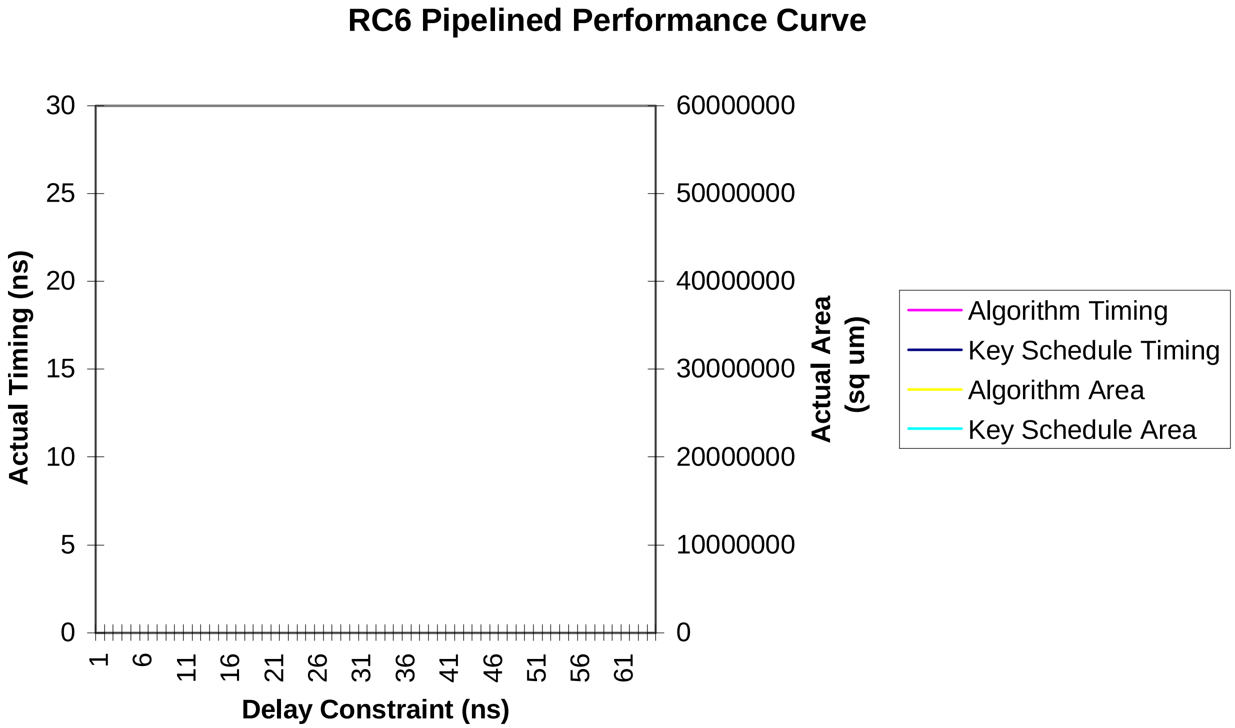


Figure 7

6.2.2.2 Key Parameters

Parameter	Iterative 3in1		Pipelined 3in1	
	Min.	Max.	Min.	Max.
Area (um2)	*	*	*	*
Transistor Count	*	*	*	*
Input/Outputs Required	520	520	520	520
Throughput (Mbps)	*	*	1192.00	2171.00
Key Setup Time Encrypt (ns)	*	*	*	*
Key Setup Time Decrypt (ns)	*	*	*	*
Algorithm Setup Time (ns)	0	0	0	0
Time to Encrypt One Block (ns)	*	*	1179.2	2146.8
Time to Decrypt One Block(ns)	*	*	1179.2	2146.8

Table 2 RC6 Summary

6.3 RIJNDAEL

6.3.1 Architecture

The following provides a high level description of the major blocks in the RIJNDAEL algorithm. Details of the components, sweeps, and their implementations can be found in the design workbook[4].

6.3.1.1 Pipelined Key Schedule

The RIJNDAEL key schedule is based on a sliding window approach as described in the algorithm specification. Multiple key sizes are based on the n-1 element and the n-k element (32 bit word organized), where k is 4,6, or 8, depending on key size. The key expansion is a linear combination of the elements, so a similar function can be used on the decrypt function to "unexpand" the keys in a reverse direction. Such an approach allows for an increase in the key agility without sacrificing significant amounts of area to store all of the expanded keys.

The encryption expansion can start immediately, with the first words of the initial key being used as expanded key. The setup time for this case is zero. During the decryption, the key is expanded to the last key, stored, and then the pipeline is run to create the previous expanded key until the last decrypt key is generated, which is the initial key. Keys are generated at a rate of four 32 bit words per round, regardless of key size, to keep up with the requirements of the algorithm block. Additional registers are used to maintain sufficient previous keys to generate the next four words of expanded key.

Keys are pulled from the bank of registers which make up the sliding window. S-Boxes are re-used, without a performance penalty, to minimize the size impact of having additional S-Boxes.

6.3.1.2 Pipelined Algorithm

The RIJNDAEL algorithm pipeline consists of a sequential mapping of the steps of the algorithm to registered stages in hardware. Each stage reflects a single round of the algorithm. The primary advantage to pipelining in this manner is the significant increase in throughput. RIJNDAEL was architected such that both the encrypt and decrypt functions could be performed with the same pipeline. This approach needed a static pipeline that could perform both functions, so the algorithm round functions contained in the package will serve a dual role by providing cases for encrypt and decrypt within the same function. The pipeline structure reflects changes in direction, such as requiring a pre-add on the encryption (first round) versus decryption requiring a post-add on the last round.

6.3.1.3 Iterative Key Schedule

The iterative version of the key schedule focuses on reducing the area of the key expansion, so only a single copy of the expansion is maintained. For encryption, as in the pipelined case, the expansion starts immediately, with no key setup required. The keys are expanded every round, producing the four 32 bit words of key required. As each new key is created and stored, the old key is overwritten.

In the case of decryption, the algorithm requires a setup time to effectively run the algorithm to the last key. This serves as the starting point for all decryptions using that key. This value will also be stored so it can be referenced on each new decryption to eliminate key setup for every new decryption.

6.3.1.4 Iterative Algorithm

The algorithm block uses the same functionality as described in the pipeline but does not re-use some of the combination functions used to construct the pipeline. Instead, the function calls are made explicitly, depending on encryption/decryption to provide the widest possible range of hardware re-use. The function calls in the encrypt and decrypt directions are not symmetrical. The algorithm processes the state data on each round, performing only one step of the algorithm per round.

6.3.2 RIJNDAEL Top Level Results

6.3.2.1 Timing and Area

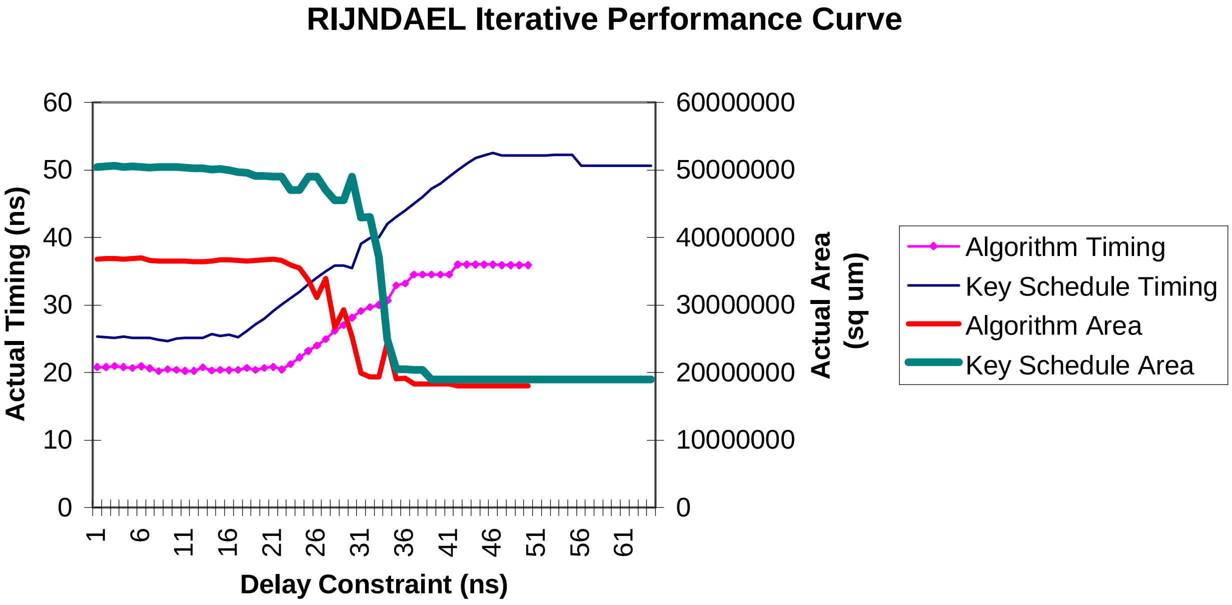


Figure 8

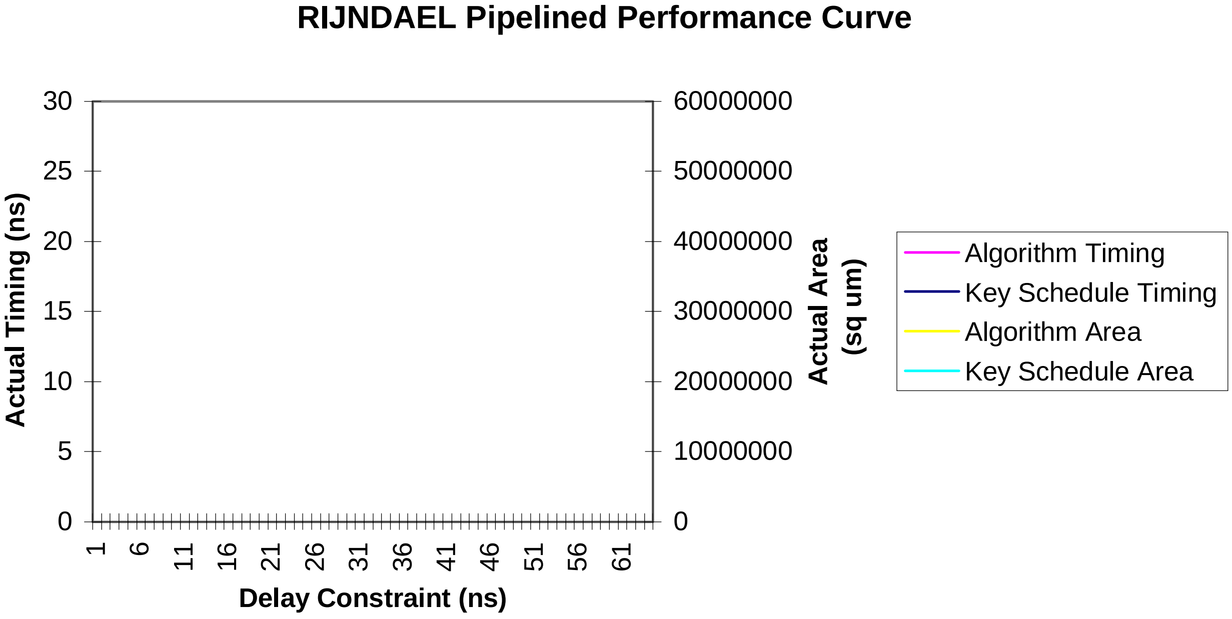


Figure 9

6.3.2.2 Key Parameters

Parameter	Iterative 3in1		Pipelined 3in1	
	Min.	Max.	Min.	Max.
Area (um2)	37034346.00	81661400.00	*	*
Transistor Count	*	*	*	*
Input/Outputs Required	520	520	520	520
Throughput (Mbps)	371.06	519.48	4060.00	5163.00
Key Setup Time Encrypt (ns)	0.00	0.00	0.00	0.00
Key Setup Time Decrypt (ns)	246.4	344.96	0	277.92
Algorithm Setup Time (ns)	0	0	0	0
Time to Encrypt One Block (ns)	493.8	346.36	247.4	346.36

Table 3 RIJNDAEL Summary

6.4 SERPENT

6.4.1 Architecture

The following provides a high level description of the major blocks in the SERPENT algorithm. Details of the components, sweeps, and their implementations can be found in the design workbook[5].

6.4.1.1 Pipelined Key Schedule

The SERPENT algorithm implements a simple expansion function for the key scheduling. The exclusive-or based function allows for quick computation and does not require key setup in the encrypt direction. Pipelining is maximized as this approach utilizes a sliding window approach, where only a small number of previous expanded keys are needed to compute the next sub-keys. However, for decryption, a key setup time is required to compute the starting point for the key expansion, which is the last set of W registers. The decrypt pipeline computes the previous set of W registers based on the current set, as the exclusive-or based expansion can be reversed easily. To save storage in this design, the keys are computed at each stage, with the decrypt case requiring a block of logic at the beginning to find the last subkeys.

The SERPENT pipelined key schedule provides two successive keys to each round of the algorithm on expansion. The algorithm will select the correct key based on the current encryption/decryption mode. The additional key allows for the rounds that require two keys to operate.

6.4.1.2 Pipelined Algorithm

The pipelined SERPENT algorithm block contains a structural model of the unraveled rounds of the algorithm. Four distinct functions are needed to implement both the encrypt and decrypt operations. The core algorithm round functions are the same for 30 rounds of the algorithm, with an internal mux/demux to select the encrypt or decrypt mode. The first two rounds of encrypt and last two rounds of decrypt distinguish the cases where the pipeline is re-routed. The encrypt will bypass the two special rounds of the decrypt while the decrypt will bypass the two special rounds of encrypt. The latency will remain the same as no extra rounds are added. The pipeline will select and re-route based on the current mode of encryption or decryption.

6.4.1.3 Iterative Key Schedule

The SERPENT iterative key schedule uses a single copy of the expansion function to generate the sub-keys, one at a time. Area can be significantly reduced using the same hardware repeatedly. Additional key setup will be required in the decrypt direction to allow for the run-up to the last key of the expansion.

6.4.1.4 Iterative Algorithm

The iterative algorithm uses the same functions as the pipeline, with the same round instance referenced repeatedly to perform the main processing of the algorithm. The special case rounds are selected by the state machine within the iterative block to determine encrypt/decrypt direction, and consequently, which pre/post add functions to perform.

6.4.2 SERPENT Top Level Results

6.4.2.1 Timing and Area

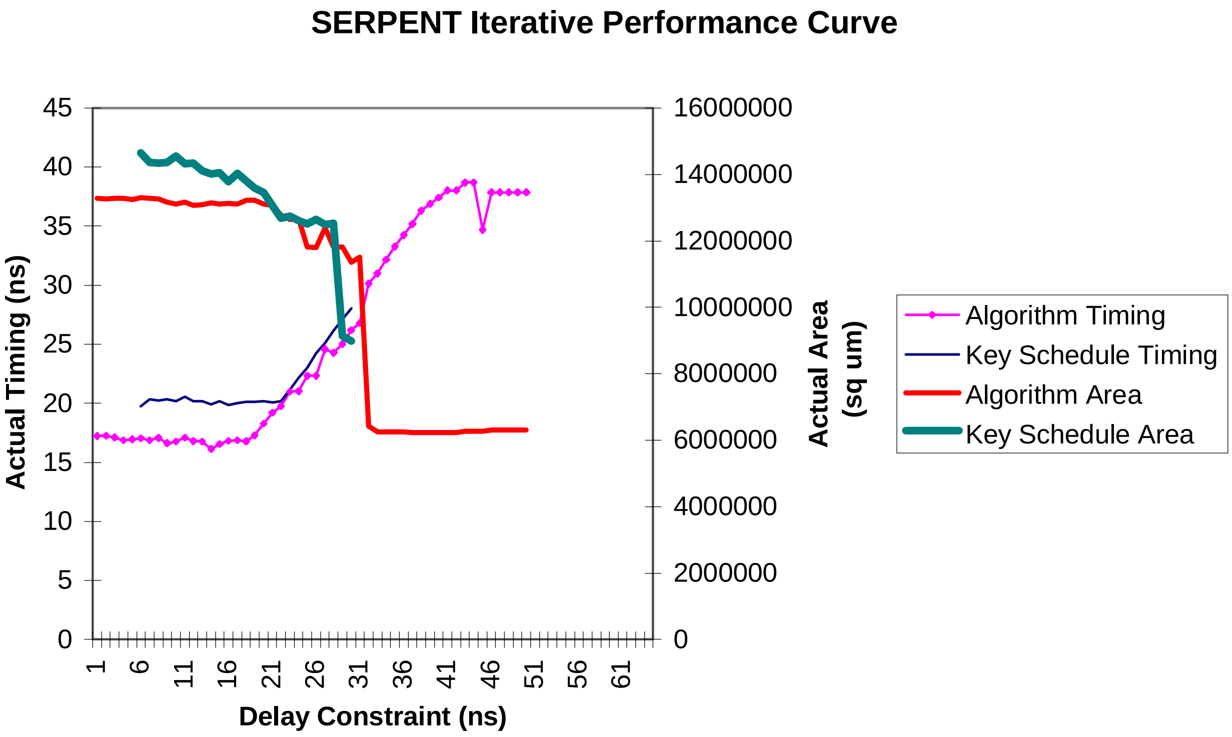


Figure 10

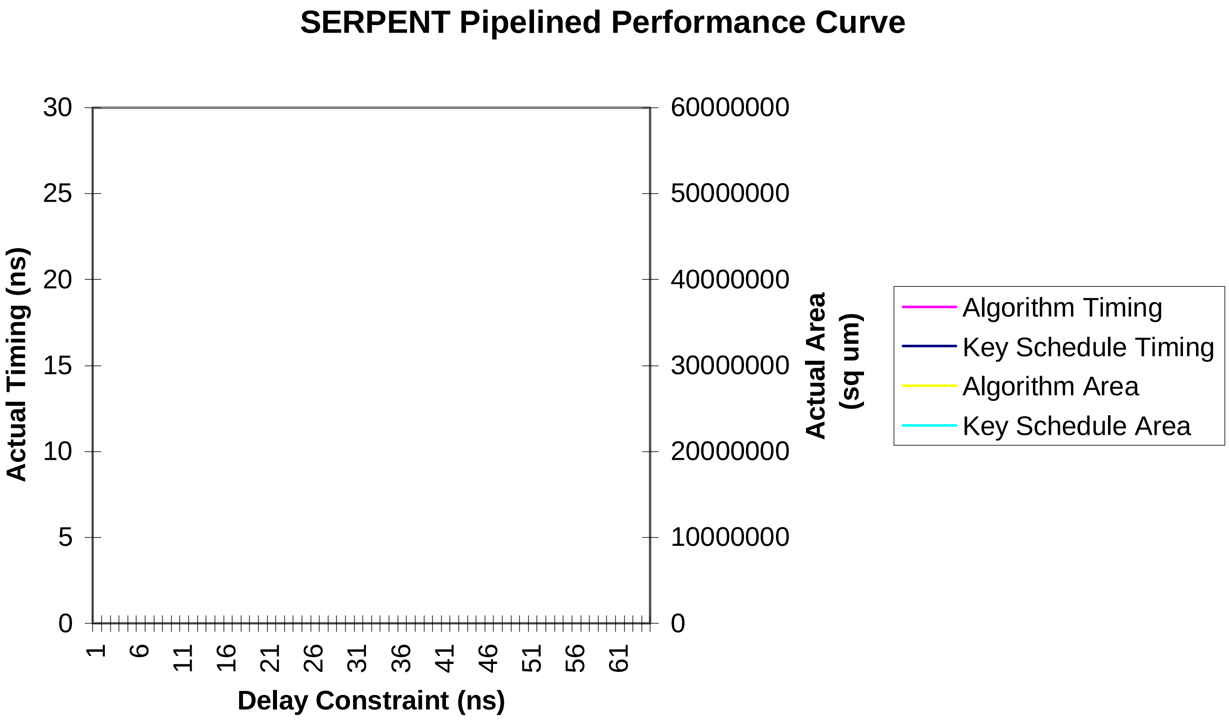


Figure 11

6.4.2.2 Key Parameters

Parameter	Iterative 3in1		Pipelined 3in1	
	Min.	Max.	Min.	Max.
Area (um2)	*	*	*	*
Transistor Count	*	*	*	*
Input/Outputs Required	520	520	520	520
Throughput (Mbps)	*	202.33	5298.01	8030.11
Key Setup Time Encrypt (ns)	19.77	*	6.74	11.76
Key Setup Time Decrypt (ns)	672.18	*	212.55	365.58
Algorithm Setup Time (ns)	0	0	0	0
Time to Encrypt One Block (ns)	632.64	*	510.08	773.12
Time to Decrypt One Block(ns)	632.64	*	510.08	773.12

Table 4 SERPENT Summary

6.5 TWOFISH

The following provides a high level description of the major blocks in the TWOFISH algorithm. Details of the components, sweeps, and their implementations can be found in the design workbook[6].

6.5.1 Architecture

A useful property of the TWOFISH architecture was the relatively large amount of re-use of design blocks. Both the Key Schedule and Algorithm utilized many of the same functions. While this does not result directly in a direct increase in performance, since key expansion and encryption are performed in parallel, it does simplify the hardware coding process. As stated, common coding techniques and processes were used for developing each algorithm design resulting in areas available for improvement in a more highly optimized design. In the case of TWOFISH, a smaller design could be created by taking advantage of the function re-use. However, as with most hardware trade-offs, this area optimization would come at the expense of performance and complex control mechanisms.

Another feature of TWOFISH is the lack of initial key runup prior to subkey expansion. In addition, the key schedule is not a feed-forward design. Each round of key schedule is independent of the previous round. This unique characteristic allowed for a Key Schedule that does not require a setup time for either encryption or decryption.

In the TWOFISH algorithm, the first step of encryption is a pre-whiten function, In hardware, this is simply an exclusive-OR. The pre-whiten step is performed during the same clock cycle as the first subkey expansion which generates the pre-whiten subkey. This was possible because the XOR function did not create a critical path concern since the main algorithm rounds incorporate an integer addition that is more complex. The result was the ability to load data and key in the same clock cycle, thereby reducing the overall time for encryption by one clock cycle.

6.5.1.1 Pipelined Key Schedule

In order to allow for either encryption or decryption, both pre-add and post-add subkeys are generated during the first pipeline stage. The post-add key is buffered through a pipelined delay until needed in the final processing step. Also, since one of the input parameters is the round number which is fixed for a given pipelined round there is an optimization or pre-calculation in each pipelined round.

6.5.1.2 Pipelined Algorithm

The algorithm is an efficient unrolling of stages because encryption and decryption are nearly identical. In addition, the symmetry allows for similar processing in both the encrypt and decrypt directions.

6.5.1.3 Iterative Key Schedule

As in the pipelined key schedule, the iterative design requires buffering of the post-add subkey until it is needed in the final processing step. However, this buffering is not required to be implemented in pipelined stages. The key schedule round is generalized such that the round number is not a fixed constant as in the pipelined case. This does not allow synthesis optimization of each round, but does save area since only one hardware block is instantiated.

6.5.1.4 Iterative Algorithm

The differences between encryption and decryption are minor such that the additional hardware to support either process in a single round adds minimal area.

6.5.2 TWOFISH Top Level Results

6.5.2.1 Timing and Area

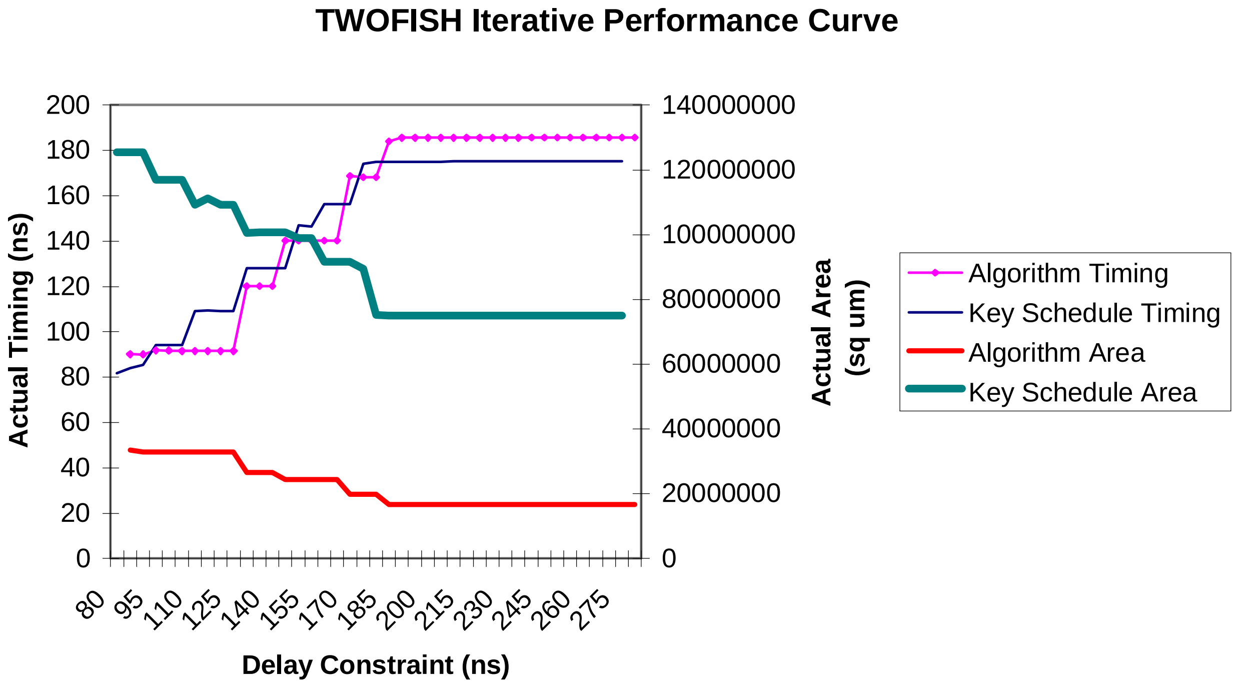


Figure 12

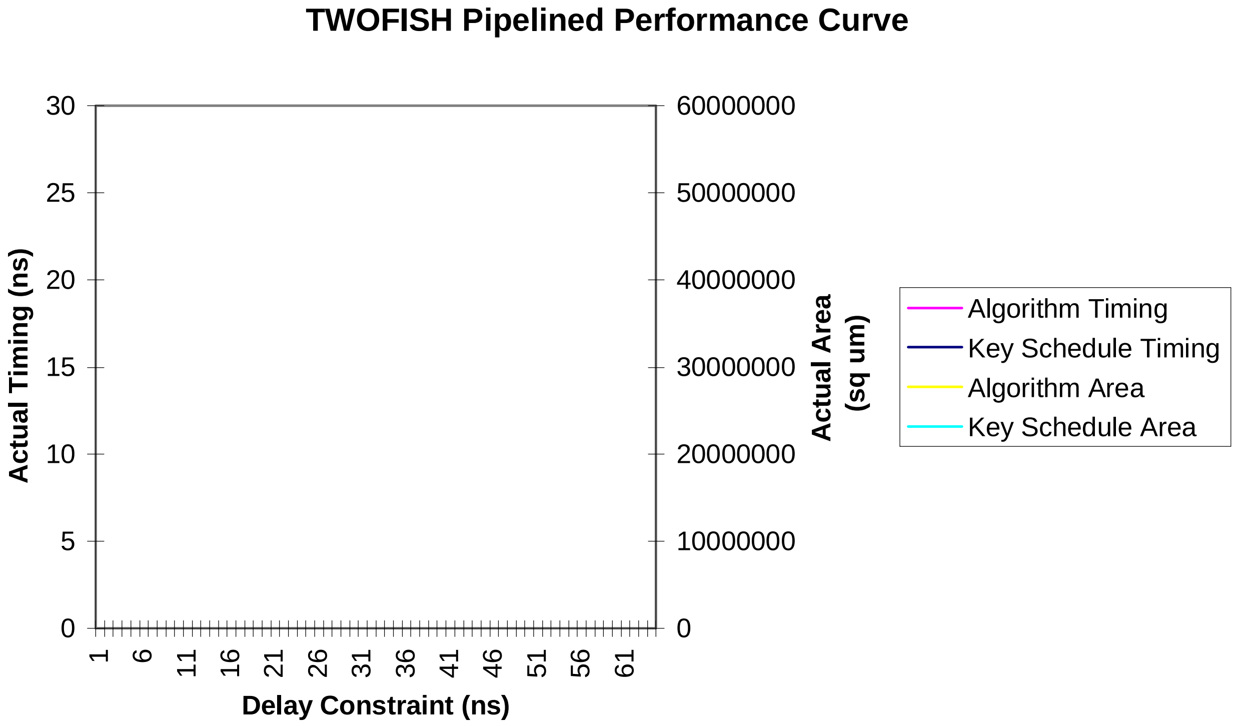


Figure 13

6.5.2.2 Key Parameters

Parameter	Iterative 3in1		Pipelined 3in1	
	Min.	Max.	Min.	Max.
Area (um2)	91686840	158300076	*	*
Transistor Count	*	*	*	*
Input/Outputs Required	520	520	520	520
Throughput (Mbps)	38.29	79.00	*	1445.55
Key Setup Time Encrypt (ns)	0	0	0	0
Key Setup Time Decrypt (ns)	0	0	0	0
Algorithm Setup Time (ns)	0	0	0	0
Time to Encrypt One Block (ns)	1620.18	3342.6	1593.9	*
Time to Decrypt One Block(ns)	1620.18	3342.6	1593.9	*

Table 5 TWOFISH Summary

7 Performance Results

A table summarizing the results and performance metrics is given below for algorithm comparison. These comparison values are given only for the combined key size implementation, which implements a selectable 128 bit, 192 bit, and 256 bit key in the same implementation.

Parameter	Algorithm				
	MARS	RIJNDAEL	RC6	SERPENT	TWOFISH
Area (um2)	*	*	*	*	*
Transistor Count	*	*	*	*	*
Input/Outputs Required	520	520	520	520	520
Throughput (Mbps)	*	519	*	202	79
Key Setup Time (ns)	*	*	*	*	*
Algorithm Setup Time (ns)	0	0	0	0	0
Time to Encrypt One Block (ns)	*	494	*	633	1620
Time to Decrypt One Block(ns)	*	494	*	633	1620

Table 6 Iterated Summary

Parameter	Algorithm				
	MARS	RIJNDAEL	RC6	SERPENT	TWOFISH
Area (um2)	*	*	*	*	*
Transistor Count	*	*	*	*	*
Input/Outputs Required	520	520	520	520	520
Throughput (Mbps)	*	5163	2171	8030	1445
Key Setup Time (ns)	*	*	*	*	*
Algorithm Setup Time (ns)	0	0	0	0	0
Time to Encrypt One Block (ns)	*	247	1179	510	1594
Time to Decrypt One Block(ns)	*	247	1179	510	1594

Table 7 Pipelined Summary

8 Summary

This paper has presented an overview of the methods and architectures used for the AES hardware comparison. The primary characteristics used for design tradeoffs in hardware engineering are area and timing. As such, each algorithm was examined from the standpoint of minimum area (iterative architecture) and maximum throughput (pipelined architecture). Further, statistics and data based on area and timing were emphasized and illustrated for each algorithm.

The results (in Section 7) show vital parameters for both the iterative and pipelined architectures of each algorithm that can be used to evaluate relative performance. The designs were not optimized for any one parameter, but rather they serve as a good testbench scoring of all the algorithms relative to one another, given the same commonly used hardware design practices and procedures. Key performance data points to highlight are minimum transistor count and maximum throughput. *

It should be emphasized that any data point based on a single parameter (e.g. transistor count or throughput) is a relatively narrow view of the algorithm's overall performance or rating. For this reason, there was no attempt to rank algorithms in order. Rather, it is left to the cryptographic community to establish a consensus of the most important parameters – in combination or alone – and to draw appropriate conclusions from the data provided herein.

* Note: Because incomplete information was available at publication time, additional results will be updated and provided to the community through NIST as the parameter information is filled in for all algorithms.

9 Acknowledgements

The authors would like to thank others at NSA for providing support to this project and in editing this report, namely Mr. Jeff Ingle. The authors also appreciate the opportunity NIST gave in encouraging this analysis, since they feel that high performance implementations for high-speed networks are an important aspect of the AES competition.

10 References

[1] W. Semancik, L. Mercer, T. Hoehn, G. Rowe, M. Smith-Luther, R. Agee, D. Fowlkes, and J. Ingle , "Cell Level Encryption for ATM Networks and Some Results from Initial Testing," , *DoD Fiber Optics Conference*, March 1994.

[2] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas, L. O'Connor, M. Peyravian, D. Safford and N. Zunic, "Mars – a candidate cipher for AES," *First Advanced Encryption Standard (AES) Conference*, Ventura, CA, 1998.

[3] R. Rivest, M. Robshaw, R. Sidney, and Y. Yin, "The RC6^{TM} Block Cipher," *First Advanced Encryption Standard (AES) Conference*, Ventura, CA, 1998.

[4] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," *First Advanced Encryption Standard (AES) Conference*, Ventura, CA, 1998.

[5] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," *First Advanced Encryption Standard (AES) Conference*, Ventura, CA, 1998.

[6] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall, "Twofish: A 128-Bit Block Cipher," *First Advanced Encryption Standard (AES) Conference*, Ventura, CA, 1998.

High-Speed MARS Hardware

Akashi Satoh[†], Nobuyuki Ooba[†], Kohji Takano[†], Edward D'Avignon[††]
[†]IBM research, Tokyo Research Laboratory, IBM Japan Ltd., 1623-14, Shimotsuruma,
Yamato-shi, Kanagawa 242-8502, Japan
{akashi, ooba, chano}@jp.ibm.com
[††]IBM Corporation, Poughkeepsie, NY 12601, USA
davignon@us.ibm.com

March 15, 2000

Abstract. High-speed MARS encryption/decryption hardware was developed using a 0.18μm IBM CMOS technology. In order to boost performance, a special adder and multiplier was designed by optimizing the adder block structure and interconnections between adder cells using signal delay profiles. A description of the hardware including block diagrams and data flow diagrams is presented. One of the most critical portions of the design is the special adder and multiplier. The design philosophy and tradeoffs used in these pieces are discussed. Finally, performance and size estimates are presented along with the rationale behind them. The design achieves 677Mbit/s data rate for encryption when using cipher block chaining and 1.28Gbit/s for decryption and other encryption modes in 13.8Kgates + 2.25Kbyte SRAM.

1. Introduction

MARS [1] is a symmetric-key block cipher, supporting 128-bit blocks and a variable key size. It is designed to take advantage of the powerful operations supported by today's computers, resulting in a much improved security/performance tradeoff over existing ciphers. We developed high-speed MARS hardware for use when additional performance or security is required over a software implementation. Since MARS uses 32-bit multiplications and additions in conjunction with S-box lookups, it is essential for MARS hardware to have a high-speed multiplier and adder. The key to realizing high-speed arithmetic circuits is to first break one operation into parallel sub-operation blocks, then precisely adjust and control the number of signal delays from each block. We developed an automatic circuit generation program, which optimizes the parallel block structure and the wiring interconnection by using the signal delay profiles. A high-speed adder with the combination of carry-skip [2] and carry-select [3] techniques designed for an RSA encryption LSI [4] was implemented in the final stage of the multiplier. These arithmetic circuits boost the speed of MARS hardware while maintaining compact silicon area.

In this paper, we first show the data path level design of the MARS hardware with an overview of the MARS algorithm and how encryption and decryption are performed. Next, we discuss the techniques that apply to the adders and multipliers to realize the high-speed MARS computation. Finally, we give estimated performance results and the size of the MARS hardware.

2. MARS Algorithm and Hardware Architecture

2.1. Hardware Block Diagram

We designed the MARS hardware entirely from the gate level to the chip level, so that it is ready for chip fabrication. Figure 1 shows the block diagram of the hardware. It has a chip external bus, which consists of a 32-bit data bus, a 10-bit address bus, four control signals, and a clock, to interface with external logic, such as a CPU. Through the bus, the external logic will read and write message data and the key. The hardware has a forward/backward mixer, a cryptographic core for MARS encryption/decryption, and a key expander for key setup. During those operations, two S-boxes and key storage are accessed. Each S-box is a 32-bit × 256-word SRAM. The key storage is a 32-bit × 64-word SRAM.

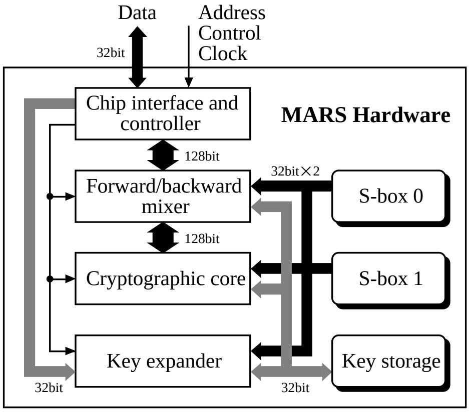
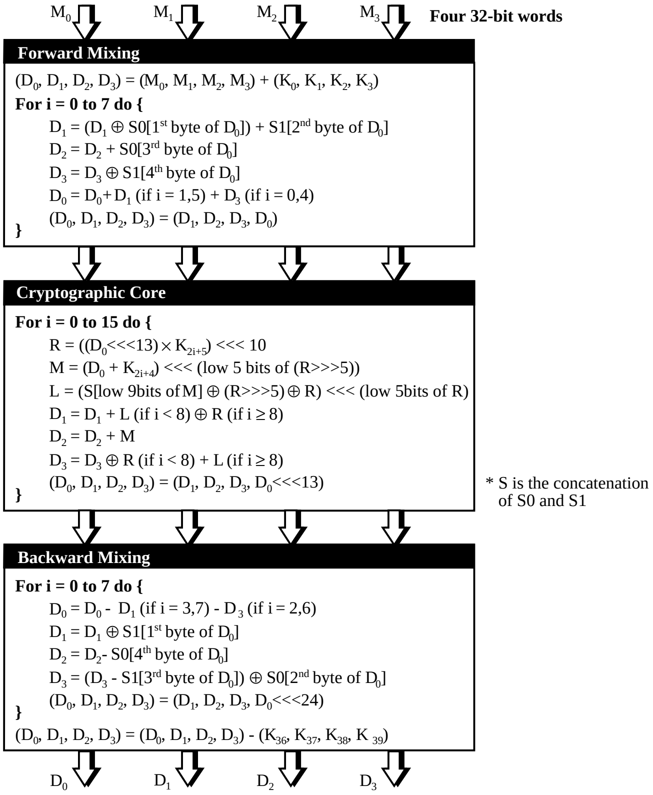


Figure 1. Block diagram of MARS hardware.

2.2. Encryption Procedure

The MARS encryption procedure has three phases: 8-round "forward mixing," 16-round "cryptographic core," and 8-round "backward mixing," as shown in Figure 2. Figure 3 shows the type-3 Feistel network structure of MARS. A 128-bit plain text block is divided into four 32-bit data words M_0, M_1, M_2, M_3, and encrypted as four words D_0, D_1, D_2 and D_3. In the figure, ⊕ denotes XOR, "<<<*n*" and ">>>*n*" denote *n*-bit cyclic left and right rotations, respectively. The lower 32 bits of the results of addition, subtraction and multiplication are used; the higher bits are discarded. MARS uses S-box (32-bit × 512-word table) lookups in the key setup, encryption, and decryption procedures. The S-box is composed of two 256-entry tables S0 (the first 256 words) and S1 (the last 256 words), used in the forward and backward mixing phases. The decryption procedure is the inverse of the encryption operation, and all circuits shown in this paper are used for both procedures by switching selectors in the data paths.



```
Forward Mixing
(D_0, D_1, D_2, D_3) = (M_0, M_1, M_2, M_3) + (K_0, K_1, K_2, K_3)
For i = 0 to 7 do {
    D_1 = (D_1 ⊕ S0[1st byte of D_0]) + S1[2nd byte of D_0]
    D_2 = D_2 + S0[3rd byte of D_0]
    D_3 = D_3 ⊕ S1[4th byte of D_0]
    D_0 = D_0+ D_1 (if i = 1,5) + D_3 (if i = 0,4)
    (D_0, D_1, D_2, D_3) = (D_1, D_2, D_3, D_0)
}

Cryptographic Core
For i = 0 to 15 do {
    R = ((D_0<<<13) × K_{2i+5}) <<< 10
    M = (D_0 + K_{2i+4}) <<< (low 5 bits of (R>>>5))
    L = (S[low 9bits of M] ⊕ (R>>>5) ⊕ R) <<< (low 5bits of R)
    D_1 = D_1 + L (if i < 8) ⊕ R (if i ≥ 8)
    D_2 = D_2 + M
    D_3 = D_3 ⊕ R (if i < 8) + L (if i ≥ 8)
    (D_0, D_1, D_2, D_3) = (D_1, D_2, D_3, D_0<<<13)
}

Backward Mixing
For i = 0 to 7 do {
    D_0 = D_0 -  D_1 (if i = 3,7) - D_3 (if i = 2,6)
    D_1 = D_1 ⊕ S1[1st byte of D_0]
    D_2 = D_2- S0[4th byte of D_0]
    D_3 = (D_3 - S1[3rd byte of D_0]) ⊕ S0[2nd byte of D_0]
    (D_0, D_1, D_2, D_3) = (D_1, D_2, D_3, D_0<<<24)
}
(D_0, D_1, D_2, D_3) = (D_0, D_1, D_2, D_3) - (K_36, K_37, K_38, K_39)
```

* S is the concatenation of S0 and S1

Figure 2. MARS encryption procedure.

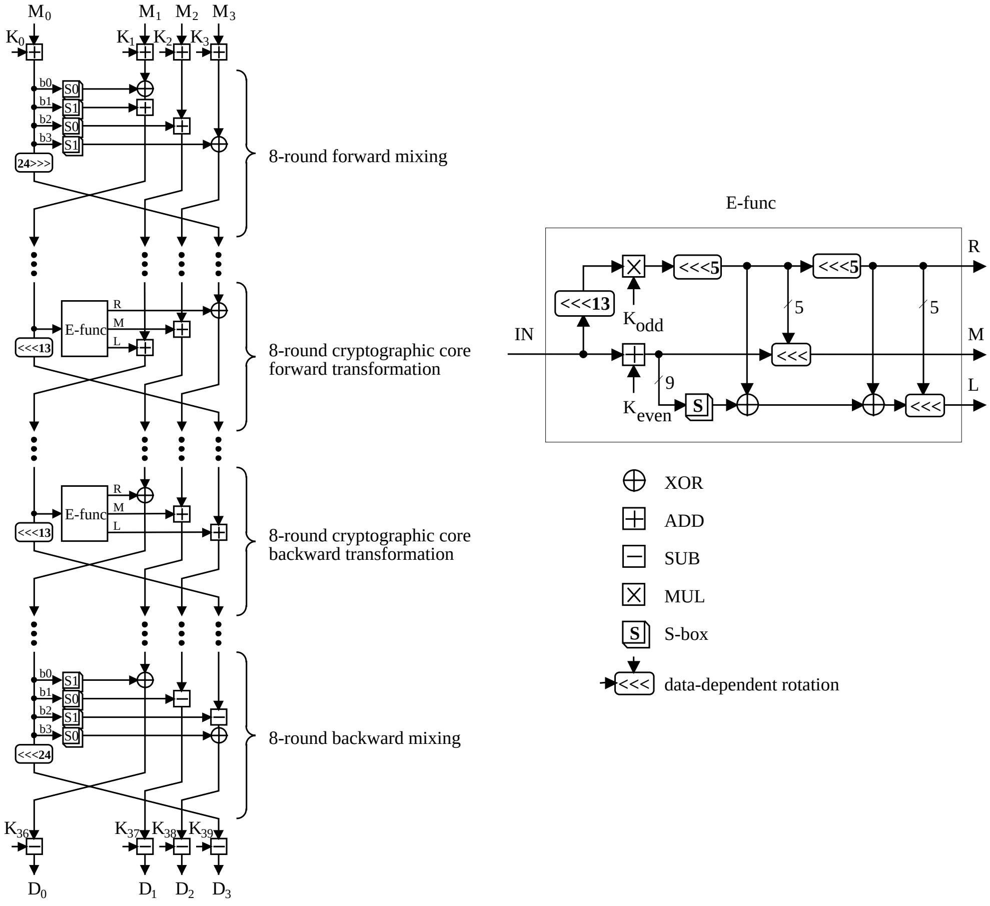


Figure 3. Type-3 Feistel network structure.

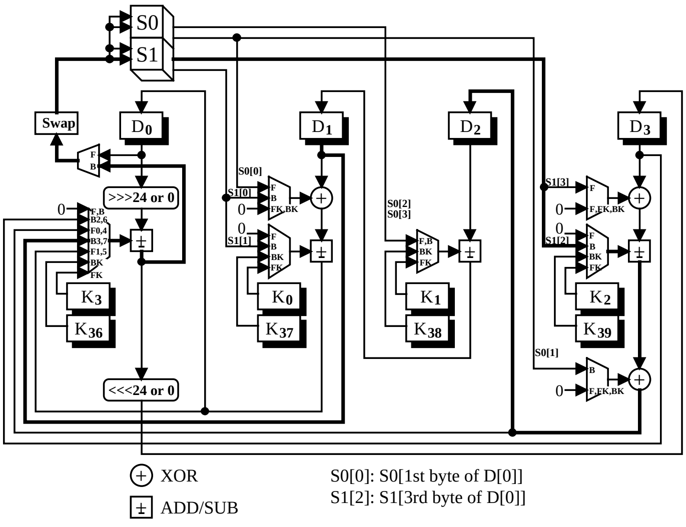


Figure 4. Forward / backward mixing data paths.

Figure 4 shows the circuit block diagram of the forward and backward mixing data paths. This circuit is also shared by the encryption and decryption procedures. Switching the selectors changes the order of the operations. The S-boxes, S0 and S1, are implemented by three-port SRAMs, one port for the write and two ports for the read operations. The thick lines show the critical path for the backward mixing process, which contains subtraction, S-box, subtraction, and XOR operations in order. Two sets of key registers $K_{0\text{-}3}$ and $K_{36\text{-}39}$ are dedicated to this mixing operation, and eight key words are copied from the 32-bit × 40-word expanded keys stored in SRAM "K".

This circuit block is used 9 times in the forward mixing mode, then one cycle is required to add the sub-keys $K_{0\text{-}3}$ to the data $D_{0\text{-}3}$, and then 8 times in the rounds of mixing operation. The backward mixing operation takes 9 cycles.

Figure 5 is the block diagram of the cryptographic core (Feistel network) data path. The thick lines specify the critical path. It consists of a multiplier, two XORs, a conditional rotator, an adder and a selector. The S-box read operation is executed in parallel with the multiplication, so that the memory access time does not affect the critical path. The S-box shares the SRAM used for the forward and backward phases shown in Figure 4. The cryptographic core operation uses this circuit in the 8-round keyed forward transformation followed by the 8-round keyed backward transformation. The cryptographic core requires 16 cycles for each 128-bit block encryption.

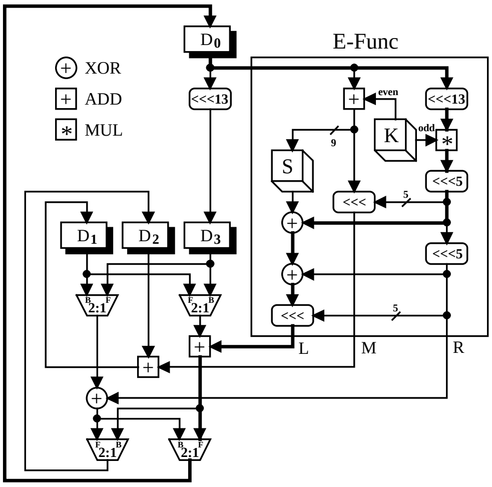


Figure 5. Cryptographic core data path.

The cryptographic core and the forward/backward mixer can operate simultaneously on separate 128-bit blocks when four-port (one for write and three for read) SRAM is used as the S-box. A 128-bit bus connection can swap data between these two circuits without additional cycles. If we share the circuits of Figure 4 with forward and backward mixing operations to save hardware resources, 18 cycles are required for one set of encryption procedures. A timing chart for this case, which is suitable for electronic codebook (ECB) encryption mode and all decryption modes, is given in Figure 6 (a). The data throughput of this architecture is 128 bits / 18 cycles. For cipher block chaining (CBC) encryption mode, the encrypted data D in the previous cycle is required before starting the current encryption of block M. In this case, the mixing phases cannot be pipelined with the cryptographic core. CBC operations require 34 cycles, with the throughput becoming 128 bits / 34 cycles. The timing chart for cipher block chaining encryption mode is shown in Figure 6 (b).

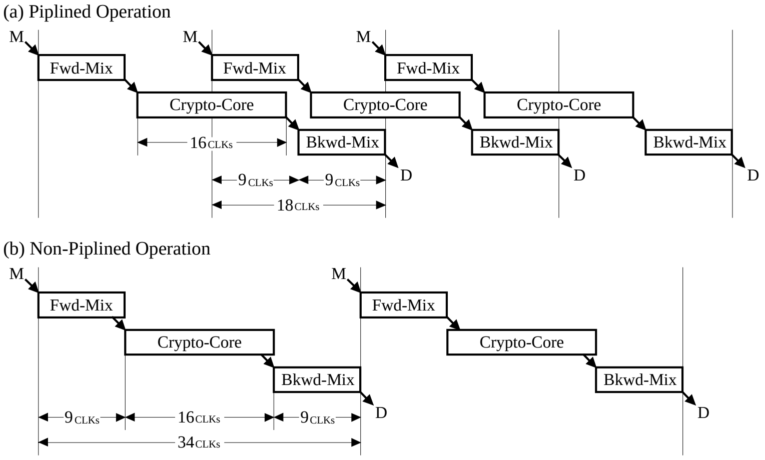
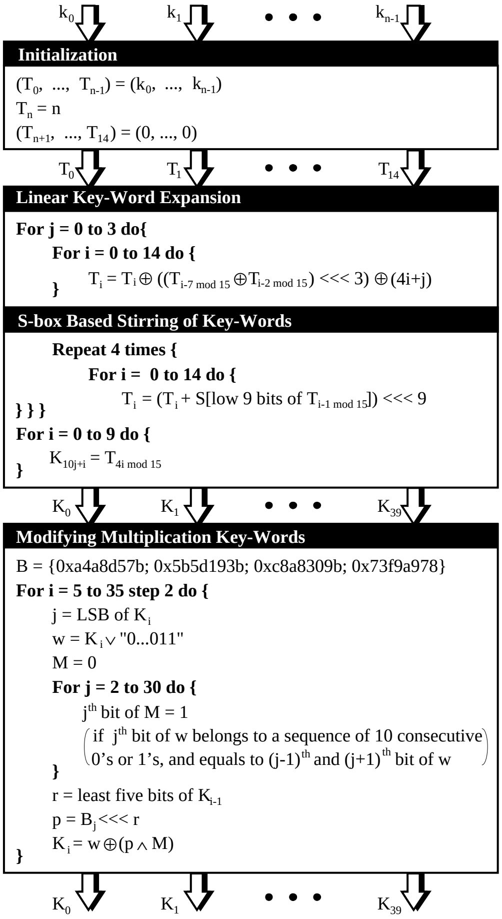


Figure 6. Timing chart of MARS encryption.

2.3. Key Expansion

The key expansion procedure, shown in Figure 7, expands the user-supplied key array, k_0, …, k_{n-1}, into a 40-word internal key array, K_0, …, K_{39}. The range of n is from 4 to 14 32 bit words, that is, MARS supports user key lengths from 128 bits to 448 bits. In the figure, bit-wise OR and AND are denoted by $\vee$ and $\wedge$, respectively. The block diagram of the key expander data paths is shown in Figure 8. The major components of the key expansion circuit are a barrel rotator, two registers, an adder, and multiplexers. The key storage "K" is implemented using a three-port SRAM. It is capable of one write and two read operations in parallel. We designed the key expander with a small number of latches in order to keep it small in size. The temporary storage T, which is used during the key expansion procedure, is implemented in the SRAM. For this reason, the key storage has 64 entries of 32 bits data. Key expansion takes 752 to 848 cycles depending on the value of the key.



Initialization

```
$(T_0, ..., T_{n-1}) = (k_0, ..., k_{n-1})$
$T_n = n$
$(T_{n+1}, ..., T_{14}) = (0, ..., 0)$
```

Linear Key-Word Expansion

```
For j = 0 to 3 do{
    For i = 0 to 14 do {
        $T_i = T_i \oplus ((T_{i-7 \bmod 15} \oplus T_{i-2 \bmod 15}) <<< 3) \oplus (4i+j)$
    }
```

S-box Based Stirring of Key-Words

```
    Repeat 4 times {
        For i =  0 to 14 do {
            $T_i = (T_i + S[\text{low 9 bits of } T_{i-1 \bmod 15}]) <<< 9$
} } }
For i = 0 to 9 do {
    $K_{10j+i} = T_{4i \bmod 15}$
}
```

Modifying Multiplication Key-Words

```
B = {0xa4a8d57b; 0x5b5d193b; 0xc8a8309b; 0x73f9a978}
For i = 5 to 35 step 2 do {
    j = LSB of $K_i$
    $w = K_i \vee$ "0...011"
    M = 0
    For j = 2 to 30 do {
        $j^{th}$ bit of M = 1
        (if $j^{th}$ bit of w belongs to a sequence of 10 consecutive
         0's or 1's, and equals to $(j-1)^{th}$ and $(j+1)^{th}$ bit of w)
    }
    r = least five bits of $K_{i-1}$
    $p = B_j <<< r$
    $K_i = w \oplus (p \wedge M)$
}
```

Figure 7. Key expansion procedure.

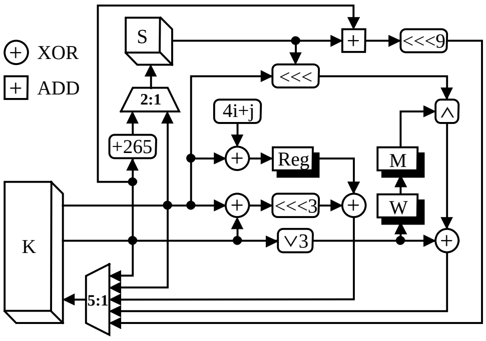


Figure 8. Key expander data path.

3. High-Speed Adder and Multiplier

3.1. High-Speed Adder

In this section, we first explain the design of a high-speed adder employing a combination of carry-skip [2] and carry-select [3] techniques used in the RSA encryption LSI [4]. This adder is used in the E-function and in the last stage of the multiplier. It is one of the most critical parts affecting MARS hardware performance.

Figure 9 shows the basic structure of the adder. It consists of ripple-carry adder blocks where each successive block is one bit longer than the block immediately below along with a carry-skip path jumping over each adder block. The delays in the ripple-carry adders and the carry-skip path are well balanced so that every carry propagates from the LSB to the MSB without waiting for the results from the other blocks. To simplify the figure, a full adder cell FA is used in the first bit of each adder block. It can be replaced by a half adder cell in the actual implementation.

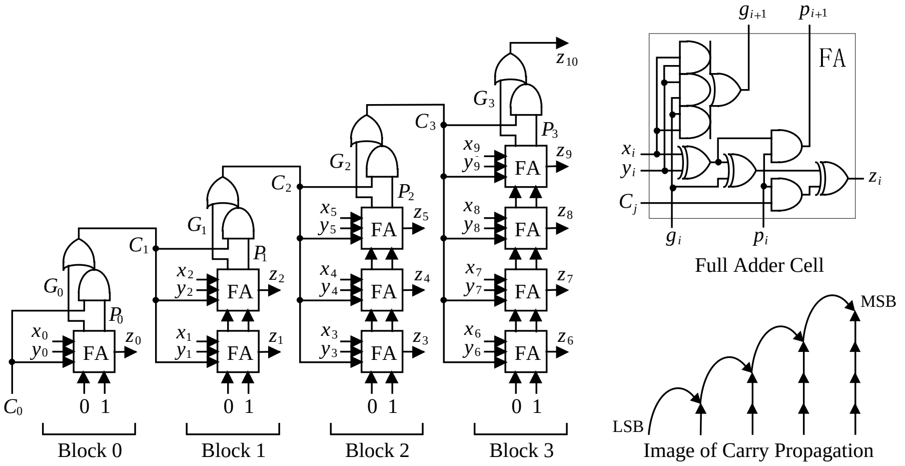


Figure 9. High-speed adder.

If two or more bits of x_i, y_i and g_i are ‘1’ in the i-th full-adder cell, carry $g_{i+1} = 1$ is generated and fed to the next cell. The cell never generates a carry if both x_i and y_i are ‘0’, regardless of the input g_i. If $g_{i+1} = 0$, either x_i or y_i is ‘0’ and the other is ‘1’, it will generate a carry if carry $C_j = 1$ comes up from the lower ripple-carry adder block j-1. For example, when $(x_3, x_4, x_5) = (1, 1, 1)$ and $(y_3, y_4, y_5) = (0, 0, 0)$, block 2 does not generate carries g_4, g_5, g_6 (= G_2). However, if the carry $C_2 = 1$ reached the block, the carry output C_3 immediately becomes ‘1.’ This means that the carry C_j can skip over the blocks one after another by pre-calculating a condition between x_i and y_i in each adder block j. The condition is defined by

$$P_j = \prod_i x_i \oplus y_i = 1, \text{ where } \oplus \text{ is XOR.}$$

By making the adder block size bigger toward the MSB side, the propagation time of P_j and C_j are equalized, and therefore the total delay time is minimized. Output z_i initially holds a sum as if the block carry C_j is 0, and is inverted by the XOR gate if $C_j = 1$ comes up later.

3.2. High-Speed Multiplier

A standard n-bit × n-bit multiplier gives a $2n$-bit result by repeatedly summing up the n-bit partial product rows. The multiplier used in MARS is not required to calculate the higher half of the result, as shown in Figure 10, so it is faster and smaller than standard multipliers. The high-speed techniques described in this section, however, can be applied to any multiplier. Figure 11 shows a Wallace tree [5], which is an adder cell array commonly used in a multiplier to reduce the number of partial product rows. The tree takes three rows and produces one carry row and one sum row, so the full adder cell, FA, is called “3:2 compressor.” This reduction is repeated until there are only two partial product rows, which are added together with a high-speed carry-propagation adder. Several tree architectures, which use 4:2, 6:2 and 9:2 compressors, were proposed [6][7] to optimize the critical path of this tree, but these compressors basically consist of 3:2 compressors. Booth encoding [8] is widely used to reduce the number of partial products, but it is a kind of 4:2 compression technique and does not change the tree structure. Oklobdzija et al [9] suggested that not all inputs and outputs from a compressor contribute equally to the delay, and the difference in using 4:2 and higher order compressors is not in the structure of the compressor but in the way they are interconnected.

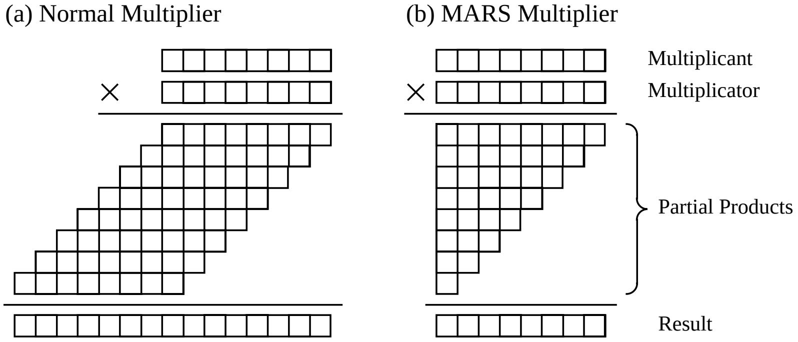


Figure 10. Partial products in MARS multiplier (*n* = 8).

In Figure 11, the input signals *x* and *y* of the full adder FA pass through two XORs to the output *s*, but the input c_i goes through only one XOR gate. The full-adder FA and half-adder HA located at the later stage of the tree are shaded in the figure. The delay profile of the tree is shown with the same shading. Here, all the XOR, NAND and AND gates are assumed to have the same propagation delay. The two signals fed into the adder at the bit-5 location come from the third-stage half adders marked with ‘*,’ but the right signal arrives earlier than the left one. In addition, the propagation delay from an input to an output varies with the types of gate and input pin locations. For example, AND usually operates faster than XOR, and NAND is faster than AND. For that reason, we developed an optimal Wallace tree generation program in consideration of six delay propagation paths of a full adder (combination of the three inputs to two outputs) and four paths of a half adder, based on a 0.18μm IBM CMOS standard cell library.

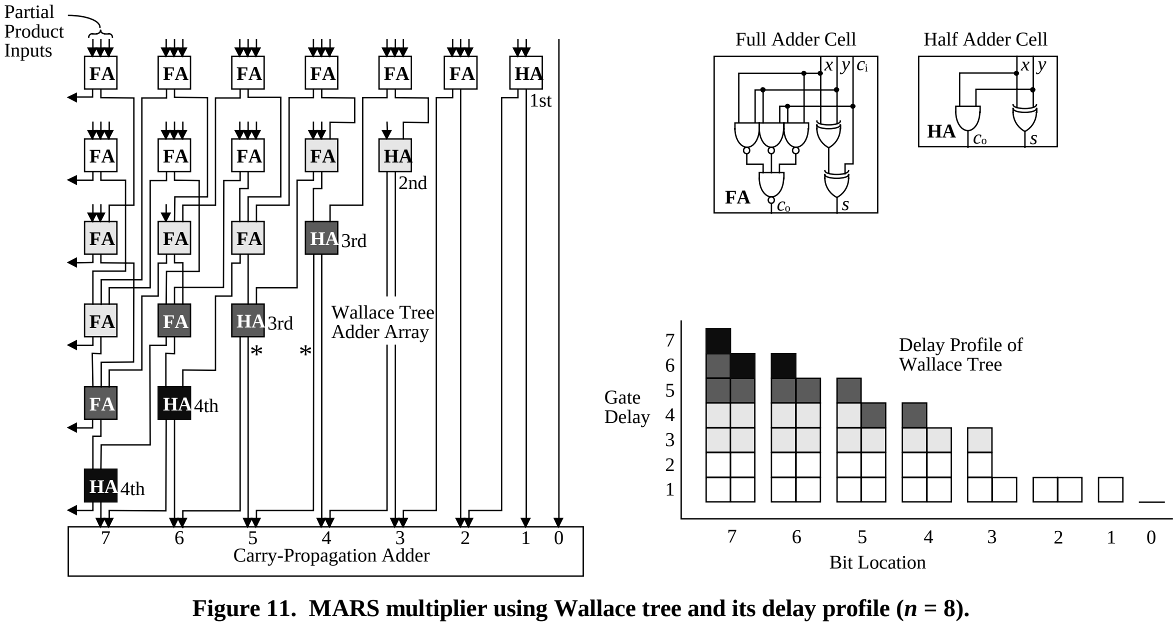


Figure 11. MARS multiplier using Wallace tree and its delay profile (*n* = 8).

Since the delay of the final carry-propagation adder is an addendum to the Wallace tree delay, the adder should have the optimized carry-propagation path for the tree delay profile. At the same time, we should consider the adder structure to determine the tree interconnection. Figure 12 shows the carry-propagation path in high-speed adders with equal and non-equal input signal arrival profiles. Both adders are identical to the one shown in Figure 9. The adder exhibits the best performance for the equal input profile (a). In case (b), the carry skipping over the adder blocks, though carry generator $C_{GEN,}$ has to wait until the carries propagates from the ripple-carry adder blocks. This is due to the slow input signals. In other words, an adder which is faster than the input delay slope is not needed. A simple ripple-carry adder can run fast enough in this case. The input signals at bit 4, 8 and 9 arrive very quickly, but these fast inputs also waste time waiting for the carry propagation from the next adder cells. To optimize performance of the multiplier, we have to make the positive delay slope gentle, and make the top of the hill as low as possible in the Wallace tree. This is achieved by optimizing the connection between the full adder and half adder gates according to their pin-to-pin internal delay profiles.

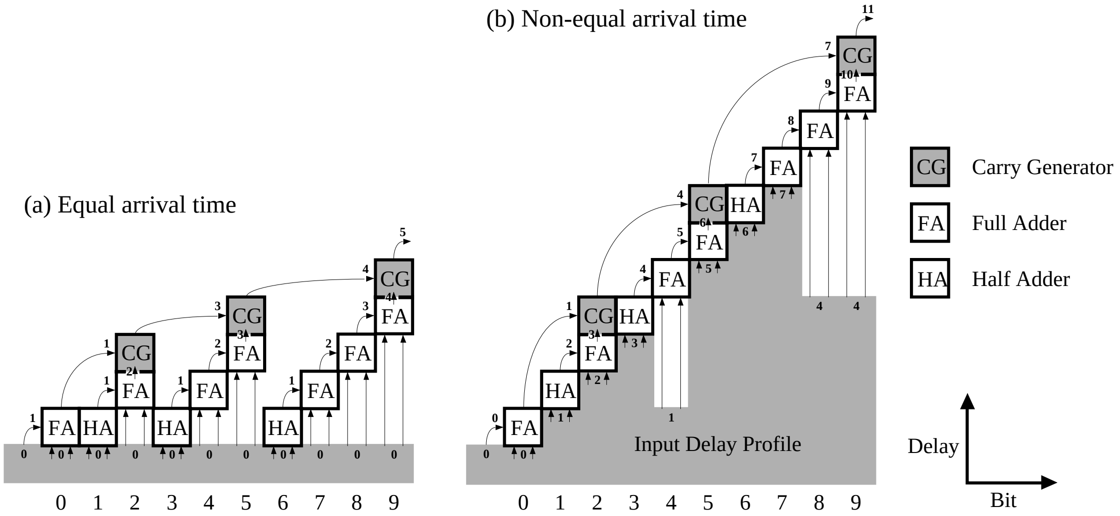


Figure 12. Carry propagation in high-speed adder on equal and non-equal input signal arrival profiles.

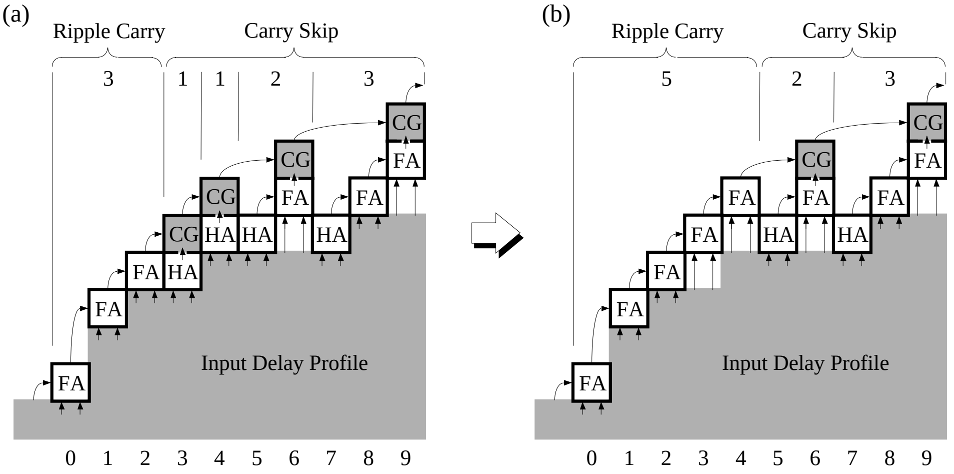


Figure 13. Adder selection over input delay profile.

Figure 13 shows an example adder structure with a positive delay slope profile. From bits 0 to 2, the slope is steeper than one FA delay, so a ripple carry adder is chosen for this part. A carry skip adder with bit blocks 1-1-2-3 is used after bit 3, in example (a). The operation of one half adder HA with a carry generator C_{GEN} is identical to that of one full adder FA, so they are replaced in example (b) to simplify the structure.

In Figures 12 and 13, the input delay time in each bit location is defined by a multiple of one adder cell delay, thus it is not difficult to optimize the adder structure. Actually, as shown in Figure 14(a), there is slack time between the input signals C and P into C_{GEN} at the 9th bit. In case (a), C is generated earlier than P, and waits for the arrival of P at C_{GEN}. When we move the location of C_{GEN} one bit left (to bit 8) so that the carry C does not waste time, the signal P has to wait instead. It is not clear which choice is better until the final adder cell is placed, and we have to choose the right combination of bit locations where C_{GEN}s are placed. In case (a), the output signal delay from the C_{GEN} is longer than that of case (b), but the carry C reaches a higher bit. We should keep the slope of the carry path over the adder blocks as gentle as possible. Therefore, the Wallace tree generator should employ a structure that has smaller value of delay/bit shown as the slope of triangles in the figure. If the structures (a) and (b) have the same slope, then we chose the former because it has higher probability to have fewer C_{GEN} cells.

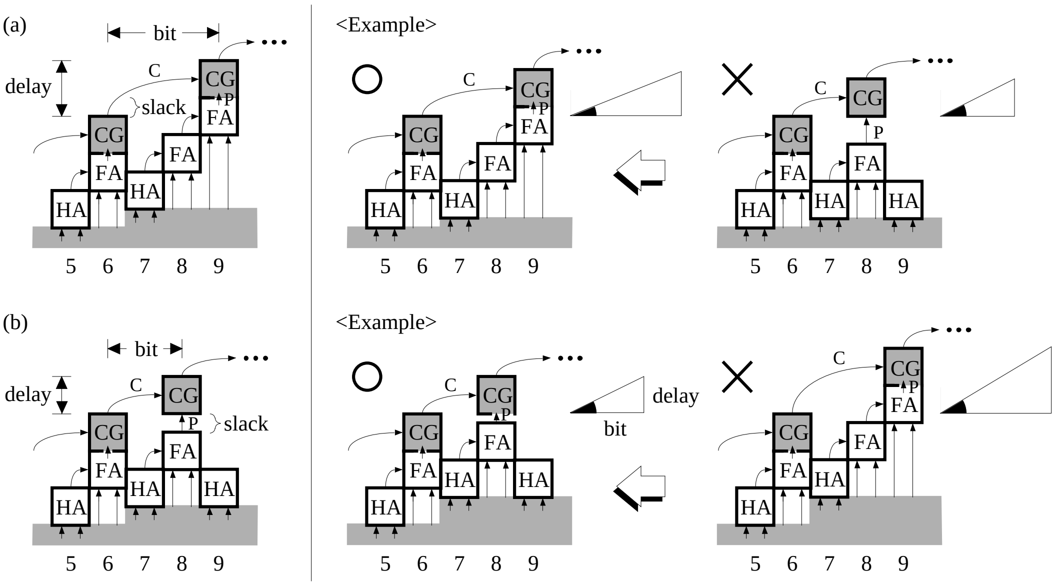


Figure 14. Carry propagation block design.

Figure 15 shows the actual delay profile of the MARS multiplier using 0.18μm IBM copper CMOS technology under nominal (V_{DD}=1.8V, Temp=25°C and L_{eff}=0.11μm) and worst case (V_{DD}=1.65V, Temp=125°C and L_{eff}=0.14μm) conditions. The output delay from the Wallace tree has an almost perfect gentle positive slope. The delay line that looks like a saw blade shows the ripple carry adder blocks. The carry skips over them smoothly. As a result, using the techniques described in this chapter realize a 2.32ns (nominal) to 3.41ns (worst) operation for the 32-bit MARS multiplier with a compact size of 3.2Kgates.

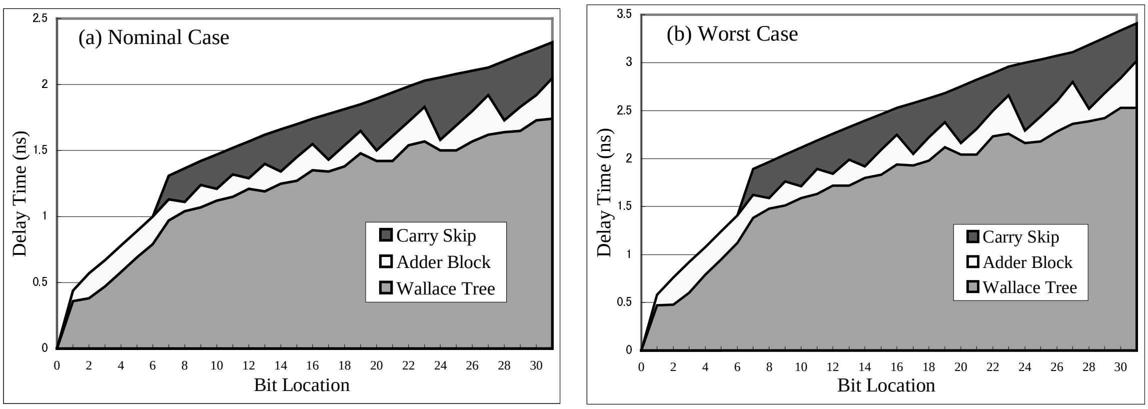


Figure 15. Actual delay profile of the MARS multiplier.

4. Performance Evaluation

We designed MARS hardware including an external interface, the control logic, and the calculation core. All the design files are written in VHDL 93. We synthesized the design using a 0.18μm IBM copper CMOS standard cell technology and evaluated its performance and size.

Table 1 shows the gate size of the logic where one gate is the size of a 2-input NAND. The memory area for the key register and S-box is shown in Table 2. We get data throughputs of 128bits / 34cycles for CBC encryption and 128bits / 18cycles for all decryption and other encryption modes, assuming a four-port SRAM implementation. However, the area of a four-port SRAM becomes larger than that of the logic part. If we do not need the high data rate, the area can be greatly reduced by using fewer-port SRAMs and a mask ROM. A two-port SRAM (one for read and one for write) for the key register halves the memory area while adding only one additional cycle for one 128-byte block encryption process. If the S-box is implemented with a single-port memory or a ROM, the cycles for the forward/backward mixing increase to 34, then the throughput of all encryption and decryption modes becomes 128 bits / 50 cycles.

The critical path delays in the forward/backward mixer and the cryptographic core under nominal and worst case conditions are shown in Figure 16. The longer delay of the cryptographic core, 5.57ns, determines the operation frequency of 180MHz (= 1 / 5.57ns) (122MHz worst case). As a result, we get a maximum data throughput of 677Mbit/s (459Mbits/s worst case) for CBC encryption and 1.28Gbit/s (867Mbits/s worst case) for other modes. All decryption modes achieve maximum throughput of 1.28Gbit/s (867Mbits/s worst case). The throughput and gate sizes for other memory implementations are summarized in Table 3.

Table 1. Logic area

Circuit Block	Gate Size
Key Expansion	2.2K
Enc/Dec Controller	4.5K
Enc/Dec Data path	6.1K
Interface + Memory Controller	1.0K
Total	13.8K

Table 2. Memory area

Function	Type	Gate Size
Key Register (256bytes)	3-port SRAM	6.8K
	2-prot SRAM	4.8K
S-box (2Kbytes)	4-port SRAM	46.2K
	3-port SRAM	30.8K
	1-port SRAM	15.4K
	ROM	6.3K

Table 3. Performance of each implementation

Memory Type		Total Gate Size (Mem+Logic)	Throughput			
Key	S-box		CBC Encryption		Other Encryption and All Decryption Modes	
			Nominal Case	Worst Case	Nominal Case	Worst Case
3-port	4-port	66.8K	677Mbit/s (34cycles)	459Mbit/s	1.28Gbit/s (18cycles)	867Mbit/s
3-port	3-port	51.4K	677Mbit/s (34cycles)	459Mbit/s	677Mbit/s (34cycles)	459Mbit/s
3-port	1-port	36.0K	460Mbit/s (50cycles)	312Mbit/s	460Mbit/s (50cycles)	312Mbit/s
2-port	1-port	34.0K	451Mbit/s (51cycles)	306Mbit/s	451Mbit/s (51cycles)	306Mbit/s
2-port	ROM*	24.9K	263Mbit/s (51cycles)	263Mbit/s	263Mbit/s (51cycles)	263Mbit/s

* 105MHz operation limited by ROM performance

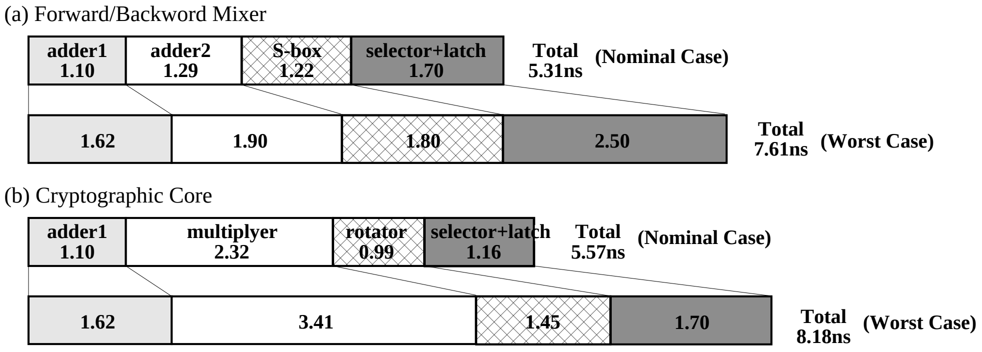


Figure 16. Critical path delay.

The technology chosen for the above estimations is a low cost copper CMOS technology several generations behind the state of the art CMOS technology. As such the performance cannot be directly compared with that of software running on today's high performance microprocessors. If built using a newer CMOS technology the performance can be expected to improve by approximately 60%.

5. Conclusion

We designed MARS hardware and estimated its size and performance. Since the MARS algorithm uses 32-bit additions and multiplications, its performance is highly dependent on the hardware design of the adder and multiplier. We designed multipliers and adders, which fully take into account the carry propagation delay. This work demonstrates that MARS can be implemented efficiently in hardware, both in terms of area and performance. We believe the design point chosen is a reasonable tradeoff of area vs. performance. We do not claim that this is the highest performance MARS design possible. Other tradeoffs may yield faster hardware implementations. Considering the size and performance in both hardware and software along with the very high security, we believe MARS is well suited to serve as the Advanced Encryption Standard algorithm.

Acknowledgement

The authors are grateful to Ms. Carolynn Burwick, Dr. Don Coppersmith, Dr. Mike Matyas, Mr. Hideto Niijima, and Mr. Nev Zunic for their reviews and comments. We also would like to thank Mrs. Akemi Ogura for supporting us generously with CAD tool operation.

References

[1] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas, L. O'Connor, M. Peyravian, D. Safford and N. Zunic: "MARS – a candidate cipher for AES," http://csrc.nist.gov/encryption/aes/round2/AESAlgs /MARS/mars.pdf, Aug. 1999.

[2] M. Lehman and N. Burla: "Skip Techniques for High-Speed Carry Propagation in Binary Arithmetic Units," IRE Trans. Elec. Comput., vol. EC-10, pp. 691-698, Dec. 1961.

[3] O. J. Bedrij: "Carry-Select Adder," IRE Trans. Elec. Comput., vol. EC-11, pp. 340-346, June 1962.

[4] A. Satoh, Y. Kobayashi, H. Niijima, N. Ooba, S. Munetoh, and S. Sone: "A High-Speed Small RSA Encryption LSI with Low Power Dissipation," LNCS 1396, pp. 174-187, 1997.

[5] C. S. Wallace: "Suggestion for a Fast Multiplier," IEEE Trans. Computers, vol. 13, no. 2, pp.14-17, Feb. 1964.

[6] A. Weinberger: "4:2 Carry Save Adder Module," IBM Technical Disclosure Bulletin, vol. 23, Jan. 1981.

[7] P. Song and G. De Michelli: “Circuit and Architecture Trade-Offs for High Speed Multiplication,” IEEE J. Solid State Circuits, vol. 26, no. 9, Sept. 1991.

[8] A. D. Booth: “A Signed Binary Multiplication Technique,” Quarterly J. Mechanical Applications in Math, vol. 4, part 2, pp. 236-240, 1951.

[9] V. G. Oklobdzija, D. Villeger and S. S. Liu: “A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach,” IEEE Trans. on Comp., vol. 35, no. 3, pp. 294-305, Mar. 1996.

Speeding up Serpent

Dag Arne Osvik [*]

March 15, 2000

Abstract

We present a method for finding efficient instruction sequences for the Serpent S-boxes. Current implementations need many registers to store temporary variables, yet the common x86 processors only have 8 registers, of which even fewer are available for computations. The instructions are also destructive, replacing one input with the output. Alternative versions of the S-box instructions are presented, requiring only 5 registers and also utilizing parallelism. Speedup of C language implementations of 24% is shown on the Pentium Pro Processor, and 42% on the Pentium, both compared to the previously fastest known implementation of Serpent.

1 Introduction

The main aspect of the finalists for the Advanced Encryption Standard is the security level they provide, especially against already known attack methods. Another aspect is the encryption speed they allow in different applications. The goal of this work has been to find ways to improve the execution speed of the Serpent algorithm on the x86 processors, including use of two-way parallel execution.

Serpent[1], being an SP-network (it consists of substitutions and permutations), has two major parts; the S-boxes and the linear transformation. The latter has a simple structure, and is well suited for manual optimization. The S-boxes are 16-element permutations, and are performed in a bit parallel (also known as bitslice) style by simple boolean operations.

2 The problem

The x86 processors, which can be found in nearly every personal computer, have some clearly distinguishing features when compared to more modern architectures. One of these is the small number of registers, only 8. Another is the instruction set, where almost all instructions always modify one of their input registers.

[*] University of Bergen, Department of Informatics, N-5020 Bergen, Norway. Email address: osvik@ii.uib.no

3 Previous work

Other efforts on optimizing Serpent have centered on the more purely mathematical problem of lowering the number of boolean operations needed to express the S-boxes [2]. Thus those essential properties of the x86 processors have been ignored. The result is a high so-called 'register pressure', meaning compilers have to put temporary variables in memory, issuing load and store instructions in addition to the actual computation. The compiler also gets the job of copying values when needed. One note is appropriate here, though; lowering the number of operations is a much better approach for RISC processors than it is for x86, as RISC instructions don't have to destroy an input value, and those processors typically have 32 registers, making register pressure a non-issue. A comparison of my results to those others (on x86) is given in a later section.

4 Our approach

One possible approach to solving a computational problem is to consider all possible computations, ordered by their length. Searching to the depth needed to find complete solutions in the case of Serpent S-boxes is infeasible using this simple approach, so we need substantial improvements.

4.1 Serpent S-boxes

The Serpent S-boxes are 16-element permutations, implying that they belong to a somewhat special subset of functions in $\{Z_{16} \rightarrow Z_{16}\}$. Now, every number from 0 to 15 can be represented by a 4-digit binary number, so these functions map 4 input bits to 4 output bits. They can also be split into 4 functions mapping 4 input bits to 1 output bit, just like any 4-bit number may be split into 4 separate bits. Now recall that any function can be uniquely specified by telling its output value for every allowed input value. In the case of 4-to-1 bit functions this is simply a list of 16 binary digits, given some ordering of the input values.

4.2 Finding solutions

We need some way to transform any 4 input bits into the corresponding 4 output bits using only those instructions available in the x86 instruction set, and in a bit parallel way. We'll use S_2 as an example:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S_2(x)$	8	6	7	9	3	12	10	15	13	1	14	4	0	11	5	2

Now rewrite x and $S_2(x)$ in binary:

x_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$S_{2,3}$	1	0	0	1	0	1	1	1	1	0	1	0	0	1	0	0
$S_{2,2}$	0	1	1	0	0	1	0	1	1	0	1	1	0	0	1	0
$S_{2,1}$	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	1
$S_{2,0}$	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0

Each column in this table contains the bits of some value for x, as well as the bits of the corresponding $S_2(x)$. The set of all columns contains all possible values for x. The number of columns is thus determined by the number of possible inputs, and is *not* related to the word length of any processor.

If we find a way of combining the x_i rows by boolean operations so that we get the $S_{2,i}$ rows, then applying those operations to the bits of an input value x is equivalent to looking up $S_2(x)$. To see how this is actually done, we will look at the execution of an instruction sequence for S_2.

The x86 instructions usable for the S-boxes are these:

Instruction	Effect	C expression
and a, b	$a := a \cdot b$	a &= b
or a,b	$a := a + b$	a |= b
xor a,b	$a := a \oplus b$	a ^= b
not a	$a := a \oplus 1$	a = ~a
mov a, b	$a := b$	a = b

Suppose we have 5 registers, named $r_0, \ldots, r_4$, available for our computations, and 4 of them initially contain our 4 input bits (r_i contains x_i, $0 \leq i \leq 3$). As r_4 is not an input register, we just ignore its previous contents. Thus we have this initial state:

r_4																
r_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
r_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
r_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
r_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The instruction sequence found by the search program (with two-way parallelism shown) is this:

mov r4, r0	and r0, r2
xor r0, r3	xor r2, r1
xor r2, r0	or r3, r4
xor r3, r1	xor r4, r2
mov r1, r3	or r3, r4
xor r3, r0	and r0, r1
xor r4, r0	xor r1, r3
xor r1, r4	not r4

Executing the first line of instructions makes the modifications $r_4 := r_0$; $r_0 := r_0 \cdot r_2$, giving us this new state:

r_4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
r_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
r_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
r_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
r_0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1

Next, we perform $r_0 := r_0 \oplus r_3$; $r_2 := r_2 \oplus r_1$.

r_4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
r_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
r_2	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
r_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
r_0	0	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0

Now things get more interesting. Notice the values in the r_2 row after $r_2 := r_2 \oplus r_0$; $r_3 := r_3 + r_4$.

r_4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
r_3	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1
r_2	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0
r_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
r_0	0	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0

r_2 is now the same as $S_{2,0}$, one of our wanted output bits.
Executing the next three lines of instruction pairs, we reach this state:

r_4	0	1	1	0	1	1	0	0	1	0	0	1	0	0	1	1
r_3	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	1
r_2	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0
r_1	0	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0
r_0	0	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0

Now r_3 is the same as $S_{2,1}$. The next two lines complete the work:

$r_4 = S_{2,3}$	1	0	0	1	0	1	1	1	1	0	1	0	0	1	0	0
$r_3 = S_{2,1}$	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	1
$r_2 = S_{2,0}$	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0
$r_1 = S_{2,2}$	0	1	1	0	0	1	0	1	1	0	1	1	0	0	1	0
r_0	0	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0

Thus we have a way of applying the function S_2, using only boolean operations with 1-bit input values. Now remember that the columns were initially just a list of possible input values. So the operations performed are actually independent of the number and contents of the columns. So we may now e.g. extend our table to 32 columns and allow any contents in each of the columns. Then, when the operations are performed, they perform S_2 32 times in parallel. This is exactly what we do on a processor with 32-bit registers.

The search for such solutions basically tries all possible instruction sequences of a given length, looking for rows equal to those of the S-box wanted. Shorter sequences are generally preferred, so we start with a small length, progressing to longer ones when no solution is found. To search for sequences capable of parallel execution, like the one above, we require that an instruction not read the output of an earlier instruction on the same line. It may write to an input register of an earlier instruction, though, as that will in no way affect the outcome of the other instruction.

4.3 Optimizations

Below are short descriptions of the most important optimizations of the search algorithm. Almost all of these avoid removing solutions without keeping an equivalent solution.

- Recursion stops when the register contents can no longer generate a permutation.
- When two instruction sequences are identified as being equivalent, we remove one of them from the search.
- No instruction other than *mov* may make a register contain a copy of the value in another register.
- Unread registers may not be written to by the *mov* instruction.
- Negated registers (those last modified by a *not* instruction) are marked as such, and may not again be negated until they have been read.
- Lookahead functions efficiently calculate a set containing all values reachable in one or two cycles.
- The search is narrowed by requiring an increasing number of result values in the registers as the search goes deeper. This constraint is important for deep searches, but its most strict variant (increasing the required number

as soon as there is at least one sequence that reached it) often drops better solutions, and should be relaxed by postponing the requirement by one or two cycles.

- The instructions are limited to using only 5 registers.

First experiences using the search program with 7 registers available showed most solutions using 6 of those, while others only used 5 registers. Further testing always provided solutions using 5 registers whenever a 6 register solution was found. Given the reduced complexity of the search, and the advantages of having the S-boxes do all their computations in only 5 registers, I chose to limit the search accordingly.

5 Results

The S-box functions chosen from the search results have the properties shown in the table. Cycle count is for running these on processors like the Pentium, with two integer execution units running in parallel.

Function	Instructions	Cycles	Registers
S_0	18	9	5
S_1	18	10	5
S_2	16	8	5
S_3	19	10	5
S_4	20	10	5
S_5	19	10	5
S_6	18	10	5
S_7	20	11	5
S_0^{-1}	19	11	5
S_1^{-1}	19	11	5
S_2^{-1}	19	10	5
S_3^{-1}	18	10	5
S_4^{-1}	20	11	5
S_5^{-1}	19	10	5
S_6^{-1}	17	9	5
S_7^{-1}	19	10	5

The low register pressure of these functions makes their compiled code completely free from loads and stores. So we only load input data and round keys, and store the result. Except for the round key loads, no memory operations are issued during encryption. This is completely different from the S-boxes used in the AES submission package[1], as well as those found by Gladman and Simpson [2], which depend heavily on memory for storage during encryption. Also, the memory footprint of the encryption routines themselves is much reduced; a fully inlined encryption requires less than 4 kilobytes.

6 Optimized implementations

Due to the problem of making C compilers schedule instructions properly for the Pentium, the S-box instructions were also incorporated into assembly routines for Serpent encryption and decryption. The result was then manually tuned for this processor (which may make it slower on other processors). The implementation was made with these constraints:

- The stack pointer register is reserved for its normal use.
- Make the routines suitable as plug-in replacements for the C routines in the AES submittal of Serpent, allowing easy testing.
- One register contains a pointer to the round key table.

Keeping the stack and key table pointers, instead of using them as general purpose registers, allows multiple simultaneous use of the routines, such as in multithreaded environments.

A new set of four round keys is loaded 33 times during an encryption or decryption. Reserving a register to point to the key table avoids having to reload the pointer every time. The ideal solution for performance is to put the round keys on the stack or in a fixed location, as that would free up the key pointer (round keys would be fetched using the stack pointer). But, since the pointer to the round key table is a parameter to the routines we replace, it is needed.

Given these limitations, we still have those five registers needed for the S-boxes, plus one free for whatever use we might have for it, like early loading of a round key. This gives the opportunity to exploit the parallelism of the Pentium nearly to its full extent, thus usually executing two instructions per cycle (some instructions can only execute one at a time). The benefit of one more free register, as could be gained by fixing the location of round keys, will thus be minimal.

7 Performance comparison

Speed testing was done on these computers:

Processor	Clock speed	RAM size	OS
486 SX	33 MHz	20 MB	Linux 2.0
Pentium	100 MHz	64 MB	Linux 2.0
(Dual) Celeron	333 MHz	256 MB	Linux 2.2

The following tables give a comparison of the different implementations of Serpent on these computers. My speed figures in Mbit/s are scaled to given clock speeds, assuming all memory operations are performed in level 1 caches. In the case of Pentium Pro, I compare against the best of Gladman's most recent numbers. On the others, numbers are compared to those reported by Granboulan [3], using Gladman's code.

- 486 DX/2-50

	Encryption		Decryption	
Implementation	Mbit/s	cycles	Mbit/s	cycles
Gladman's code	0.48	12900		
Osvik	3.8	1650	3.8	1660

- Pentium 90

	Encryption		Decryption	
Implementation	Mbit/s	cycles	Mbit/s	cycles
AES submission	7.17	1605	5.88	1956
Gladman's code	8.56	1290		
Osvik	12.7	907	12.7	905
Osvik, asm	14.4	800		

- Pentium Pro 200

	Encryption		Decryption		Key setup
Implementation	Mbit/s	cycles	Mbit/s	cycles	cycles
AES submission	21.8	1170	20.6	1301	
Gladman	27.0	945	26.9	951	1290
Osvik	33.7	759	33.2	770	1106

The compiler used to compile both my own and the AES submission C code is PentiumGCC version 2.95.2. For my own code, I used the options "-O -mpentium -fPIC -fomit-frame-pointer" on Pentium and "-O2 -mpentium -fPIC -fomit-frame-pointer" on PPro. For the AES submission code I used "-O -mpentium". Other optimization settings I tried reduced the speed achieved. All times are measured including parameter passing, function call and return from the function. Timings on the 486 are not nearly as accurate as the others, as it does not have a cycle counter.

Note: the figures quoted above are for Gladman's results in C using a static array of round keys which frees up an extra register. This only allows multiple concurrent encryptions when they all use the same key. His C++ code, which does not have this limitation, shows a 3% performance reduction.

8 Future directions

- My implementations may be further tuned - actually, I expected the Pentium assembly implementation to come close to 735 cycles for encryption.

While trying to manually optimize the encryption, I found the Pentium to be very touchy regarding tight dependencies involving rotation instructions. Given the Pentium processor's slowdown when executing such instruction sequences, 735 cycles seems to be unreachable. Still, faster S-boxes might exist, as my search has not been exhaustive.

- The key setup function can generate the encryption code with round keys embedded directly in the instructions, thus removing the load instructions and saving upto 66 cycles on the Pentium. This will increase key setup time, though.
- 3-way parallelism on x86 (AMD Athlon). This only requires a (theoretically) simple extension of my current search program. The curious can quite easily verify that S_6^{-1} and S_7^{-1} both can execute in 7 cycles with up to 3 instructions/cycle, as opposed to 9 and 10 cycles on Pentium/.../Pentium III.
- Hardware implementations have a natural emphasis on parallelism. Preliminary results in this area look extremely promising; given 3-input nand and nor gates, and (at most) 2-input versions of other gates, all S-boxes can be performed with a gate depth of only 3. Combined with a depth of 4 for the linear transformation and 1 for key mixing, this indicates that several Gb/s should be possible in CBC mode with common technology. If we can also add 3-input (n)xor, the gate depth of one round is reduced to no more than 5.
- The instruction sets of RISC processors may be viewed as a set of gates from which we can build wide S-box functions. Their lack of 3-input logical operations raises the maximum gate depth needed to 4. That is, given enough parallelism on a RISC (or EPIC) chip, all S-boxes have solutions requiring no more than 4 cycles to execute. This hardware-style RISC optimization will be further investigated in the near future.

9 Acknowledgements

I would like to thank my supervisor, Lars R. Knudsen, for proposing this project, and for his advice and criticisms regarding this article.

My parents and some of my friends and fellow students have been helpful in various ways. Most of the search and timing tests where performed on Odd Egil Nerland's computers. Gisle Sælensminde has made an Ada implementation using my S-box functions, and in the process he made a Python script automating inlining of the encryption functions. This was necessary to avoid stressing the GCC register allocator with these rather intricate S-boxes. My C and assembly implementions were also made using slightly modified versions of his script, saving much work and time.

References

[1] RJ Anderson, E Biham, LR Knudsen, “Serpent: A Proposal for the Advanced Encryption Standard”

[2] BR Gladman:
http://www.btinternet.com/~brian.gladman/cryptography_technology/

[3] L Granboulan:
http://www.dmi.ens.fr/~granboul/recherche/AES/timings.html

[4] Intel Corporation, “Intel Architecture Optimization Manual”, Order Number 242816-003, 1997.

Appendix

Below are all the S-box functions selected from the search results. The functions expect their input values to be in r0 .. r3, ordered from least to most significant bit. The contents of r4 are ignored. Output values are given in the registers listed at the bottom of each table, again ordered from least to most significant bit.

S_0		S_0^{-1}	
r3 ^= r0	r4 = r1	r2 =~ r2	r4 = r1
r1 &= r3	r4 ^= r2	r1 |= r0	r4 =~ r4
r1 ^= r0	r0 |= r3	r1 ^= r2	r2 |= r4
r0 ^= r4	r4 ^= r3	r1 ^= r3	r0 ^= r4
r3 ^= r2	r2 |= r1	r2 ^= r0	r0 &= r3
r2 ^= r4	r4 =~ r4	r4 ^= r0	r0 |= r1
r4 |= r1	r1 ^= r3	r0 ^= r2	r3 ^= r4
r1 ^= r4	r3 |= r0	r2 ^= r1	r3 ^= r0
r1 ^= r3	r4 ^= r3	r3 ^= r1	
		r2 &= r3	
		r4 ^= r2	
r1, r4, r2, r0		r0, r4, r1, r3	

S_1		S_1^{-1}	
r0 =˜ r0	r2 =˜ r2	r4 = r1	r1 ˆ= r3
r4 = r0	r0 &= r1	r3 &= r1	r4 ˆ= r2
r2 ˆ= r0	r0 |= r3	r3 ˆ= r0	r0 |= r1
r3 ˆ= r2	r1 ˆ= r0	r2 ˆ= r3	r0 ˆ= r4
r0 ˆ= r4	r4 |= r1	r0 |= r2	r1 ˆ= r3
r1 ˆ= r3	r2 |= r0	r0 ˆ= r1	r1 |= r3
r2 &= r4	r0 ˆ= r1	r1 ˆ= r0	r4 =˜ r4
r1 &= r2		r4 ˆ= r1	r1 |= r0
r1 ˆ= r0	r0 &= r2	r1 ˆ= r0	
r0 ˆ= r4		r1 |= r4	
		r3 ˆ= r1	
r2, r0, r3, r1		r4, r0, r3, r2	

S_2		S_2^{-1}	
r4 = r0	r0 &= r2	r2 ˆ= r3	r3 ˆ= r0
r0 ˆ= r3	r2 ˆ= r1	r4 = r3	r3 &= r2
r2 ˆ= r0	r3 |= r4	r3 ˆ= r1	r1 |= r2
r3 ˆ= r1	r4 ˆ= r2	r1 ˆ= r4	r4 &= r3
r1 = r3	r3 |= r4	r2 ˆ= r3	r4 &= r0
r3 ˆ= r0	r0 &= r1	r4 ˆ= r2	r2 &= r1
r4 ˆ= r0	r1 ˆ= r3	r2 |= r0	r3 =˜ r3
r1 ˆ= r4	r4 =˜ r4	r2 ˆ= r3	r0 ˆ= r3
		r0 &= r1	r3 ˆ= r4
		r3 ˆ= r0	
r2, r3, r1, r4		r1, r4, r2, r3	

S_3		S_3^{-1}	
r4 = r0	r0 |= r3	r4 = r2	r2 ˆ= r1
r3 ˆ= r1	r1 &= r4	r0 ˆ= r2	r4 &= r2
r4 ˆ= r2	r2 ˆ= r3	r4 ˆ= r0	r0 &= r1
r3 &= r0	r4 |= r1	r1 ˆ= r3	r3 |= r4
r3 ˆ= r4	r0 ˆ= r1	r2 ˆ= r3	r0 ˆ= r3
r4 &= r0	r1 ˆ= r3	r1 ˆ= r4	r3 &= r2
r4 ˆ= r2	r1 |= r0	r3 ˆ= r1	r1 ˆ= r0
r1 ˆ= r2	r0 ˆ= r3	r1 |= r2	r0 ˆ= r3
r2 = r1	r1 |= r3	r1 ˆ= r4	
r1 ˆ= r0		r0 ˆ= r1	
r1, r2, r3, r4		r2, r1, r3, r0	

S_4						S_4^{-1}					
r1	^=	r3	r3	=~	r3	r4	=	r2	r2	&=	r3
r2	^=	r3	r3	^=	r0	r2	^=	r1	r1	|=	r3
r4	=	r1	r1	&=	r3	r1	&=	r0	r4	^=	r2
r1	^=	r2	r4	^=	r3	r4	^=	r1	r1	&=	r2
r0	^=	r4	r2	&=	r4	r0	=~	r0	r3	^=	r4
r2	^=	r0	r0	&=	r1	r1	^=	r3	r3	&=	r0
r3	^=	r0	r4	|=	r1	r3	^=	r2	r0	^=	r1
r4	^=	r0	r0	|=	r3	r2	&=	r0	r3	^=	r0
r0	^=	r2	r2	&=	r3	r2	^=	r4			
r0	=~	r0	r4	^=	r2	r2	|=	r3	r3	^=	r0
						r2	^=	r1			
r1, r4, r0, r3						r0, r3, r2, r4					

S_5						S_5^{-1}					
r0	^=	r1	r1	^=	r3	r1	=~	r1	r4	=	r3
r3	=~	r3	r4	=	r1	r2	^=	r1	r3	|=	r0
r1	&=	r0	r2	^=	r3	r3	^=	r2	r2	|=	r1
r1	^=	r2	r2	|=	r4	r2	&=	r0	r4	^=	r3
r4	^=	r3	r3	&=	r1	r2	^=	r4	r4	|=	r0
r3	^=	r0	r4	^=	r1	r4	^=	r1	r1	&=	r2
r4	^=	r2	r2	^=	r0	r1	^=	r3	r4	^=	r2
r0	&=	r3	r2	=~	r2	r3	&=	r4	r4	^=	r1
r0	^=	r4	r4	|=	r3	r3	^=	r4	r4	=~	r4
r2	^=	r4				r3	^=	r0			
r1, r3, r0, r2						r1, r4, r3, r2					

S_6						S_6^{-1}					
r2	=~	r2	r4	=	r3	r0	^=	r2	r4	=	r2
r3	&=	r0	r0	^=	r4	r2	&=	r0	r4	^=	r3
r3	^=	r2	r2	|=	r4	r2	=~	r2	r3	^=	r1
r1	^=	r3	r2	^=	r0	r2	^=	r3	r4	|=	r0
r0	|=	r1	r2	^=	r1	r0	^=	r2	r3	^=	r4
r4	^=	r0	r0	|=	r3	r4	^=	r1	r1	&=	r3
r0	^=	r2	r4	^=	r3	r1	^=	r0	r0	^=	r3
r4	^=	r0	r3	=~	r3	r0	|=	r2	r3	^=	r1
r2	&=	r4				r4	^=	r0			
r2	^=	r3									
r0, r1, r4, r2						r1, r2, r4, r3					

S_7						S_7^{-1}					
r4	=	r1	r1	|=	r2	r4	=	r2	r2	^=	r0
r1	^=	r3	r4	^=	r2	r0	&=	r3	r4	|=	r3
r2	^=	r1	r3	|=	r4	r2	=~	r2	r3	^=	r1
r3	&=	r0	r4	^=	r2	r1	|=	r0	r0	^=	r2
r3	^=	r1	r1	|=	r4	r2	&=	r4	r3	&=	r4
r1	^=	r0	r0	|=	r4	r1	^=	r2	r2	^=	r0
r0	^=	r2	r1	^=	r4	r0	|=	r2	r4	^=	r1
r2	^=	r1	r1	&=	r0	r0	^=	r3	r3	^=	r4
r1	^=	r4	r2	=~	r2	r4	|=	r0	r3	^=	r2
r2	|=	r0				r4	^=	r2			
r4	^=	r2									
r4, r3, r1, r0						r3, r0, r1, r4					

Session 8:

"Algorithm Submitter Presentations"

Submitter Statements

IBM Comments

Third AES Conference
April 13, 2000

**Don Coppersmith, Rosario Gennaro, Shai Halevi, Charanjit Jutla,
Stephen M. Matyas Jr., Mohammad Peyravian, David Safford, Nevenko Zunic**

Introduction:

All five of the AES finalist candidates are solid ciphers, with no known weaknesses. It seems likely that any of the candidates would make a good standard. In this short paper, we summarize some qualitative and objective comments on the finalists, and make recommendations for final selection.

General Comments on the Candidates

MARS

MARS has one of the widest security margins, both in terms of number of rounds, and in terms of diversity (as its security relies on a combination of several different "strong operations" and on a heterogeneous structure). MARS is the only candidate with a heterogeneous structure, which was a deliberate design feature to help resist unknown attacks. Also, the design of the round function in MARS lends itself to analysis. In particular, a nearly complete characterization is known for the differential behavior of the round function, and independent analysis has been published.

At the same time, MARS is also a very fast cipher. In fact, in some of the measurements, MARS posted the fastest C and Java benchmarks. In Gladman's C benchmarks, MARS average performance across all key sizes was second only to RC6.

One concern raised about MARS was that it was hard to implement on memory constrained environments. In response to this criticism, the key schedule was tweaked prior to round 2, significantly reducing memory requirements.

Another criticism raised about MARS concerned its complexity. We feel that this was partly due to our extremely detailed presentation and analysis of the algorithm. We subsequently released a simplified description including simplified pseudocode which fits on a single page, (which is included later in this paper). In addition, using implementation lines as a complexity measurement, MARS is ***less*** complex than Twofish, Rijndael, and Serpent.

RC6

RC6 has a simple, elegant round function, and it is the fastest cipher in speed tests. A possible concern about RC6 is that its round function may be "too simple". Specifically, the combination of multiplications and rotation, although providing some excellent properties, is a "single point of failure" in RC6 (as it does not use S−boxes). Also, RC6 seems to have the lowest security margin of the candidates in terms of number of rounds.

Rijndael

Rijndael is a fast cipher, which is very flexible for implementation. It is important to note that its speed on 256 bit keys is lower than MARS or Twofish.

Rijndael has a round function which is hard to analyze, and a key schedule that makes it easier to mount power attacks. Also, the fact that the round function can be expressed as only a few simple algebraic operations makes one wary of potential algebraic attacks against it.

The structure of Rijndael and Square is new, and not fully understood. In "The Block Cipher Square", Daemen, Knudsen, and Rijmen presented an attack unique to the Square structure, which caused them to increase the number of rounds. The existence of attacks unique to Square call into question Rijndael's long term resistance.

Rijndael's mode with only 10 rounds has a relatively low security margin.

Serpent

Serpent has very wide security margins in terms of number of rounds, and very strong mixing. On the down side, it is quite slow, and it also has a key schedule that makes power attacks easier to mount. As there are other candidates with good security margins, and much faster performance, we feel that Serpent is too slow.

Twofish

Twofish is a flexible cipher in terms of implementation tradeoffs, and it is also one of the fastest ciphers (except for its key−schedule). It has good security margins, and reasonable complexity.

A concern about Twofish is that it is very hard to analyze its security. Its round function was engineered to provide flexibility, rather than to facilitate analysis. Indeed, although a lot of effort has already been invested in its analysis, it is safe to say that the exact properties of the round function are not very well understood. Moreover, the reliance on key dependent S−boxes which are not generated pseudorandomly, makes the analysis even harder.

Another drawback of the key dependent S−boxes is that they are inherently more costly. In Twofish this extra cost can be shifted between the key−setup and the cipher, but nonetheless it is always there. Finally, the key schedule of Twofish makes power attacks easier, since the entire key can be deduced from only the initial whitening key.

Complexity/Size of the Candidates

As mentioned earlier, MARS is actually not a complex algorithm. One way to measure complexity is to count lines needed to implement the cipher. Here are some measurements of Gladman's C code implementations, which can be used to compare complexity:

Cipher	Lines	LOC	Statements
RC6	116	71	86
MARS	424	298	249
Twofish	496	346	224
Rijndael	449	282	212
Serpent	623	479	620

(*Lines* counts the lines in the implementation, including comments and blanks; *LOC* (lines of code) counts only lines with statements, and *statements* counts the number of C statements.) As expected, RC6 is significantly simpler. Surprisingly, Serpent is significantly more complex to implement. MARS, Twofish, and Rijndael fall in the middle, with comparable complexity. In addition, to show the conceptual simplicity of MARS, here is the entire pseudocode for MARS encryption in 30 lines, (counting comments and blank lines).

```
// Forward Mixing
(A,B,C,D) = (A,B,C,D) + (K0,K1,K2,K3)
For i = 0 to 7 {
  B = (B ^ S0[A]) + S1[A>>>8]
  C = C + S0[A>>>16]
  D = D ^ S1[A>>>24]
  A = (A>>>24) + B(if i=1,5) + D(if i=0,4)
 (A,B,C,D) = (B,C,D,A)
}

// Keyed Transformation and E−Function
For i = 0 to 15 {
  R = ((A<<<13) * K[2i+5]) <<< 10
  M = (A + K[2i+4]) <<< (low 5 bits of (R>>>5))
  L = (S[M] ^ (R>>>5) ^ R) <<< (low 5 bits of R)
  B = B + L(if i<8) ^ R(if i>=8)
  C = C + M
  D = D ^R(if i<8) + L(if i>=8)
 (A,B,C,D) = (B,C,D,A<<<13)
}

// Backward Mixing
For i = 0 to 7 {
  A = A − B(if i=3,7) − D(if i=2,6)
  B = B ^ S1[A]
  C = C − S0[A<<<8]
  D = (D − S1[A<<<16]) ^ S0[A<<<24]
 (A,B,C,D) = (B,C,D,A<<<24)
}
(A,B,C,D) = (A,B,C,D) − (K36,K37,K38,K39)
```

Performance, Complexity, and Relative Security Margin

In this section we have collected and summarized some measurements of performance, and complexity, and estimates of security margin. For performance, we use Gladman's C code results [1]. Note that Rijndael's performance varies based on key size. While other papers have analyzed the candidates on other platforms, only performance on the NIST selected reference platform has received adequate analysis and review, so we use those numbers here.

As a simple complexity measurement, we count lines in Gladman's C implementations [2]. As these have all been written by the same person to the same API, with the same style, the line counts indicate relative complexity. For security margin, we use Biham's analysis [3] of rounds divided by minimum secure rounds, to get a ratio, in which large numbers represent higher (better) margins.

Cipher	Speed(Mb/sec)	Setup(Clocks)	Lines	Security Margin
RC6	94.2	1875	116	**1.0**
Mars	69.4	2134	424	1.6
Twofish	68.8	**8493−15616**	496	1.6
Rijndael	50.5−70.3	207−1983	449	1.3−1.8
Serpent	**26.7**	1296	**623**	1.9

In this table, we have **highlighted** values that are less competitive compared to the other candidates. This table makes clear the tradeoffs between speed and margins. RC6 is fastest, with the lowest margin. Serpent is slowest with the highest margin. The Serpent code is surprisingly more complex than the others, while RC6 is, as expected, the simplest code, with the others comparable between the extremes.

Recommendation Summary

RC6 is an elegant, fast, and well analyzed cipher, and would normally be considered the obvious best candidate, but for a standard that is supposed to last twenty years, its security margin is perhaps a bit too close to the edge. If only one candidate is chosen, RC6 is perhaps a bit risky.

Of the other ciphers, Serpent is too slow. Rijndael's structure is new and less well understood, and it has a slight disadvantage in performance with large keys. The security of Twofish is difficult to analyze, given its key dependent S−box, and it has a slight disadvantage in key setup performance. Since MARS has well understood and analyzed components, has a solid security margin, is fast, and does not have the large key or key setup performance problems, it is the best choice.

Should two candidates be selected, we feel that RC6 would be the obvious second choice, since the risk from its low margin would be much less of an issue, given the existence of the other cipher to fall back on. Its simplicity and tiny size make it very easy to add as a second cipher to any implementation.

References:

1. http://www.btinternet.com/~brian.gladman/cryptography_technology/aes2/ index.html
2. http://www.btinternet.com/~brian.gladman/cryptography_technology/aes2/aes.r2.algs.zip
3. http://www.cs.technion.ac.il/~biham/Reports/aes−comparing−revised.ps.gz

RC6 as the AES

Ronald L. Rivest[1], M.J.B. Robshaw[2], and Yiqun Lisa Yin[3]

[1] M.I.T. Laboratory for Computer Science, 545 Technology Square,
Cambridge, MA 02139, USA. rivest@mit.edu
[2] 88 Hadyn Pk. Rd., London, W12 9AG, UK. mrobshaw@supanet.com
[3] NTT Multimedia Communications Laboratories, 250 Cambridge Ave.,
Palo Alto, CA 94306, USA. yiqun@nttmcl.com

Introduction

After more than a year of design and nearly two years of scrutiny, the process to choose the Advanced Encryption Standard is drawing to a close. We are now left with five designs that would each be a good choice as the final AES. These five ciphers have radically different design philosophies and they have very different security and performance properties. No one cipher sticks out as being the natural choice in all respects.

During the design of RC6 our pragmatic aim was to satisfy as many goals as possible while keeping the cipher simple. Only by keeping a cipher simple can one achieve a well-understood level of security, good performance, and a versatility of design that makes the cipher highly adaptable to future demands.

We believe that we have been successful in this approach and developments over the last two years have only served to strengthen our views. We believe that RC6 would make an excellent choice as the final AES.

Security through simplicity

Despite the talk of "margins for security" and "fair" or "minimal" round assessments, the most important measure of the likely security of a cipher is quite simply the amount of scrutiny it has received. Yet it is not clear how much attention the different ciphers have received. Cryptanalysts have full-time jobs teaching in a university or working on a range of unrelated industry projects. Very few, if any, will have looked at more than two finalists in any depth, let alone all five.

A simple cipher is one that is easily described and readily remembered. It will, as a direct result, be analyzed and scrutinized widely [2, 4, 5, 8, 11]. Not only will it receive the greatest quantity of analysis - it will also receive the most accurate analysis. During the design of RC6 we performed what we believe to be one of the most accurate assessments of the security of any of the AES finalists [4]. RC6 is not so complicated that approximating models have to be introduced (as with MARS [3] and Twofish [17]). Instead we were able to get a remarkably accurate view of the strength offered by RC6 using direct analysis[4]. In this way we were

[4] Since it is easy to define simplified and small block-size variants of RC6, the cryptanalyst can perform far more extensive analysis and experimentation.

able to make a careful decision on how many rounds RC6 should have so that we delivered good performance once our security goals had been attained. In the case of some finalists new attacks have improved on the work of the designers. Yet it is a vindication of our approach that when other techniques are applied, as was done by Knudsen and Meier [11] (and also Baudron et al. [2]), they give surprisingly similar results to those provided by our own analysis. This isn't a "small margin for security". Rather it is a carefully assessed, and remarkably accurate margin for security.

As well as being earned, some faith in a cipher can be inherited. The time for assessment of the finalists throughout the AES process has been a little less than two years. By building on the knowledge of earlier ciphers we gain insight into the security of a new cipher. Clearly RC6 was designed in the light of experience gained with perhaps the most studied modern cipher, RC5 [14]. And not only with regards to the structure of the round function. We decided to choose a key schedule for RC6 that was identical to that for RC5. No other AES finalist uses a key schedule that has been open to public analysis for nearly six years. Given the problems some finalists have in the key schedule, either with key separation in the case of Twofish [12] or with related-key attacks in the case of Rijndael [7], this is a very important attribute.

The AES effort is so important that we should not be relying on crude and subjective metrics for our decisions. The process of subtracting some arbitrary number of rounds from the number of proposed rounds - arbitrary numbers that might in one case be taken from the designers documentation and in another from direct independent analysis - can be a misleading way of comparing the AES finalists. To quote [18]: "These comparisons are fundamentally flawed, because they unfairly benefit algorithms that have been cryptanalyzed the least." Instead, the true security of a cipher depends on

- the amount of cryptanalytic scrutiny received,
- the accuracy of existing cryptanalysis,
- the ease with which verifying experiments can be conducted on a cipher,
- the amount of earlier cryptanalytic analytic work that can be used in the assessment of the cipher, and,
- the accuracy of the designers initial estimates.

We believe that on all counts RC6 is most suited to be chosen as the AES.

Performance through simplicity

Most of today's high-end computing base is deployed in PC's either in the workplace or at home, and these are 32-bit machines. Here RC6 typically offers exemplary performance. Some restricted devices that are currently quite widely deployed are 8-bit based. These might include a relatively insignificant fraction of mobile devices, but would most likely be smart cards. However, when we couple the needs of greater processing power with the inevitable drop in prices of 32-bit processors, it is very clear that the mobile computing device market, including smart card applications, will inevitably shift to a 32-bit oriented processor base. This trend may take a few years to come to fruition, but its results are likely to be with us for the 20 or 30 years that might be required for the AES.

With regards to very cheap smart-cards with old 8-bit processors, it has already been observed [9] that such very cheap smart-cards are vulnerable to system attacks and are inherently insecure. Such insecurities would apply to any of the AES finalists. As a result we should be careful that we do not place too much weight on the performance of a cipher in an environment that is both insecure now and obsolete (perhaps even non-existent) in a few years time. Nevertheless such processors are currently deployed and the AES may well be desired in such applications. The first question we should ask is whether performance is an important issue in such situations? What applications are going to be used on such cheap 8-bit smart cards? Certainly they won't require bulk encryption - at most a few blocks of data will be processed. So, the performance of any of the five AES finalists is going to be adequate.

On a separate issue it is repeatedly claimed (almost to the point of folklore and most surprisingly in [18]) that an implementation of RC6 requires at least 176 bytes of RAM. Yet Keating [10] has already shown that this is not the case and that RC6 can be implemented in around 120 bytes of RAM. So we can conclude that all the AES finalists can be implemented, and can be expected to offer adequate performance, on cheap (insecure) low-end smart cards.

Looking to future architectures, fine-grained estimates today of performance on future architectures really don't seem to be terribly useful. Technology evolves in unpredictable ways (for instance the growing significance of DSPs) and it seems likely that technology will evolve to best support whichever of the AES finalists is chosen. Instead, experience in the area of 32-bit processors shows that there is nothing intrinsically unsuitable about any of the five finalists for future architectures and future designs can be expected to devote significant support to providing the best possible performance from the final AES.

We provide some additional observations.

- Hand-optimized assembly code will offer the best algorithm performance on any processor. Yet often, developers will use portable code in a higher-level language and compile it for the environment of use. Under such circumstances the simplicity of a cipher is very important since it allows a compiler to produce well-optimized code. This means that good performance can be achieved without time-consuming and costly hand optimizations or lengthy code that tries to choose among a dozen different optimization strategies.
- The simplicity of a cipher is most acutely reflected in the Java performance of a cipher. This is in terms of code-size, performance, and potentially most critically, the amount of dynamic RAM used during the encryption process. With the increased importance of the Internet and its extension to mobile devices, the performance of the finalist in Java could well be vital. While there may well be many small processors in the coming years [18] many of them will in fact be Java-based, for instance in set-top boxes.

– One possible future trend is the growth of the market [13] for DSPs and/or microprocessors with DSP capability. RC6 not only performs very well on processors of this type [19], but gains its impressive performance without look-up tables which provide additional burdens on memory requirements.

We believe that excellent performance of RC6 on 32-bit processors, the close convergence in performance between simple compiled code and hand-optimized assembly, and outstanding performance in Java and in DSP environments, all make RC6 ideally suited to be chosen as the AES.

Versatility through simplicity

One of the early stated aims of the AES process was that the final cipher be "simple and versatile". For RC6 these were design goals.

RC6 is fully parameterized; the number of encryption rounds, the size of the encryption key (not just the three must-support values of 128, 192, and 256 bits), and the block-size can all be easily and readily changed. This kind of flexibility is an integral design feature. For most of the other finalists it is not at all clear how a change to the block size, or the use of an extremely long encryption key, would be accommodated.

These could be important considerations. For some applications, a developer may wish to call on a 64-bit block cipher perhaps as a drop-in replacement to DES. With RC6 as the AES, such a variant is readily described. At the other extreme, it is possible that in the near future a 256-bit hash value will be preferred. The most natural way to do this when using an AES candidate as the basis for a hash function would be to change the block-size.

As another example of the flexibility of RC6, the key schedule allows for very long keys (for example up to 1024 bits) to be used without a compromise to performance. This is not that important for encryption, but it does provide extraordinary improvements to the performance of the Davies-Meyer hashing mode [16]; potentially to the point of providing hashing performance comparable to that offered by dedicated hash functions.

Simplicity and versatility go hand-in-hand. Once again, we believe that RC6 would be the most suited finalist to become the AES.

Conclusions

The three most important attributes of the final AES are security, performance, and versatility. With RC6 we achieve all three goals. RC6 is so simple that the full details of the cipher can be recalled at will. Through simplicity we have developed a truly versatile cipher. We have also developed a cipher that offers exceptional performance, and gives the best all-round suitability in Java with all the implications this holds for future applications. Most importantly, though, existing analysis on RC6 is not only by far the most extensive of any of the finalists, it is also the most accurate and the most detailed.

For these reasons we believe that RC6 is ideally suited to be the final AES.

References

1. R. Anderson, E. Biham, and L.R. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard.
2. O. Baudron, H. Gilbert, L. Granboulan, H. Handschuh, A. Joux, P. Nguyen, F. Noilhan, D. Pointcheval, T. Pornin, G. Poupard, J. Stern and S. Vaudenay. Report on the AES candidates. In Proceedings of *The Second AES Candidate Conference*, pages 53–67. March 22-23, 1999.
3. C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. Matyas, L. O'Conner, M. Peyravian, D. Safford, and N. Zunic. MARS - a candidate cipher for AES. June 10, 1998.
4. S. Contini, R.L. Rivest, M.J.B. Robshaw, and Y.L. Yin. The security of RC6. Available from `www.rsasecurity.com/rsalabs/aes/`.
5. S. Contini, R.L. Rivest, M.J.B. Robshaw, and Y.L. Yin. Improved analysis of some simplified variants of RC6. In L. Knudsen, editor, Fast Software Encryption, Lecture Notes in Computer Science Volume 1626, pages 1-15, Springer-Verlag, 1999.
6. J. Daemen and V. Rijmen. AES Proposal: Rijndael. June 11, 1998.
7. N. Ferguson, J. Kelsey, B. Schneier, M. Stay, D. Wagner, and D. Whiting. Improved cryptanalysis of Rijndael. Preprint.
8. H. Gilbert, H. Handschuh, A. Joux, and S. Vaudenay. A statistical attack on RC6. Preprint.
9. S. Halevi. Suggested "tweaks" for the MARS cipher. Submitted to NIST at the end of Round 1 evaluation. Available via `csrc.nist.gov`.
10. G. Keating. Performance analysis of AES candidates on the 6805 CPU core. In Proceedings of *The Second AES Candidate Conference*, pages 109–114. March 22-23, 1999. Available from `www.ozemail.com.au/ geoffk/aes-6805`.
11. L.R. Knudsen and W. Meier. Correlations in RC6. Preprint.
12. F. Mirza and S. Murphy. An observation on the key schedule of Twofish. Proceedings of the Second AES Candidate Conference, pages 151-154.
13. O. Port. Chips for the post-PC era. *Business Week*, Annual Special Issue, page 96, March 27, 2000.
14. R.L. Rivest. The RC5 encryption algorithm. In B. Preneel, editor, Fast Software Encryption, Lecture Notes in Computer Science Volume 1008, pages 86-96, Springer-Verlag, 1995. Available from `theory.lcs.mit.edu:80/~rivest/`.
15. R.L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6 Block Cipher. v1.1, August 20, 1998. Available from `www.rsasecurity.com/rsalabs/aes/`
16. M.J.B. Robshaw. Hashing with the AES finalists. Preprint.
17. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Twofish: A 128-bit Block Cipher. 15 June, 1998.
18. B. Schneier and D. Whiting. A performance comparison of the five AES finalists. Preprint.
19. T. Wollinger, M. Wang, J. Guajardo, and C. Paar. How well are high-end DSPs suited for the AES algorithms? Preprint.

This article was processed using the LaTeX macro package with LLNCS style

Rijndael for AES

Joan Daemen, Proton World, daemen.j@protonworld.com
Vincent Rijmen, KULeuven, vincent.rijmen@esat.kuleuven.ac.be

1. Introduction

In this document we give a short overview of the reasons why Rijndael should be selected as the AES. We have divided our arguments into four categories:

- **Security:** Rijndael has the same objective security level as the other finalists, and can easily be implemented in a secure way.
- **Efficiency:** Rijndael has a large "performance margin" compared to the other candidates.
- **Design philosophy:** The clear design has many advantages: easy implementable on a wide range of platforms, easy to get confidence in the claimed security level, ...
- **Extensions:** Rijndael is easily extendable to other key and block lengths.

Finally, we discuss the issue of multiple AES algorithms.

2. Security

2.1 Objectively demonstrable security

Until now, for none of the 5 AES finalists, an attack has been published that demonstrates a weakness inherent in the design. Hence, from a cryptanalytical point of view, all 5 ciphers are equivalent.

2.2 Suitability for secure implementation

In software, Rijndael can be implemented using the operations bitwise XOR, table-lookup and 8-bit shifts. Serpent requires no table-lookups but more general shifts and rotations and bitwise boolean operations.

Twofish additionally requires 32-bit addition and both MARS and RC6 even require 32-bit multiplication and shifts over data-dependent off-sets. The presence of these operations makes the latter three algorithms harder to implement in a secure way on smart cards [DaRi99].

2.3 Adding rounds

For all well-designed block cipher, the complexity of published cryptanalytic attacks increases with the number of rounds in the cipher. This has already been taken into account in the Rijndael design: the increasing number of rounds for increasing key lengths assures a growing security marging against cryptanalytic attacks.

In fact, the number of rounds is a parameter that can be increased further, *without a need for any additional specifications*. In applications where the confidence in Rijndael's security doesn't match the importance of the confidentiality/integrity, or in the hypothetical case that an effective attack on Rijndael would be published, a Rijndael version with an increased number of rounds can be used.

3. Relative efficiency

The relative efficiency of the different finalists can be shown by comparing optimal implementations on several platforms. Given the fact that the different design teams have taken different security margins, the question rises how to compare the algorithms on equal footing. One approach is to determine a minimum number of rounds that has to be used in order to resist currently known attacks, and to add some rounds extra [Bi99]. Unfortunately, not all ciphers have been subjected to the same amount of study. Furthermore, there is no consensus on how many rounds one should add to get an adequate security margin. For instance, how should the added security of an extra round of a (generalised) Feistel cipher be compared with a round of an S-P-network ?

On the other hand, the performance of all the algorithms has been evaluated on many different platforms, and all algorithms got their fair share of attention. Therefore we propose to compare the other AES finalists to Rijndael variants with an adapted number of rounds, such that both algorithms execute in the same time. In Table 1 we list for each AES finalist the number of Rijndael rounds (including the implied round key generation) that can be executed in the same time. Nominally, Rijndael has 10 rounds (for 128-bit keys).

We consider the following platforms:

- **Pentium II/Pro:** representative processor for PCs today;
- **Motorola 6805:** representative processor for smart cards today.

Moreover, we give numbers for different amounts of data treated with the same key:

- **many blocks:** indicative if the same key is used for a considerable amount of data (say at least some Kbytes).
- **4 blocks:** indicative if AES is used to secure a small amount of data. In most financial transactions the amount of data that is subject to a MAC is indeed below 64 bytes. This includes electronic purse, debit/credit and ticketing transactions that will be used in timing-critical applications such as public transport and toll-road payment automation.
- **1 block:** relevant if AES is used as the compression function of a hash function, for PIN code encipherment/decipherment or for session/instance key derivation (in smart card, terminal and/or Host security module) typical for payment systems.

Processor	# blocks	source	DES***	MARS	RC6	Serpent	Twofish
Pentium II/Pro	many	[Li00]	-	13	9	38	12*
	4	[Co99]	-	28	15	33	27
	1	[Co99]	-	46	22	36	25
Motorola 6805**	many	[Ke99]	30	30	28	110	23
	4	[Ke99]	32	52	45	107	23
	1	[Ke99]	37	114	91	100	22

Table 1 Number of rounds in Rijndael, given the same number of cycles

* The Twofish design team measures the performance of Twofish with code that has the used key compiled into the executable. We use the code by Aoki and Lipmaa, slightly slower than the self-modifying (!) code by the Twofish team.

****** For Twofish, only the results of the designers are available. For MARS and RC6 we use the implementations for smartcards with massive RAM available. For Rijndael, we average cipher and inverse cipher speed.

******* For DES, the number of blocks is doubled as the block length is only 64 bits

4. Design philosophy

In the Rijndael design, we have tried to keep everything as simple as possible. Complexity has been added only when necessary to thwart attacks. One example is the key schedule, that is very simple and efficient compared to that of other AES finalists.

Other “simplicity” properties include:

- Symmetry in the round transformation and across the rounds,
- Orthogonality of components,
- Absence of arithmetic operations.

These properties lead to a number of advantages that are treated in the following sections.

MARS and Twofish, on the contrary, have both a very complex round function, with many different operations. According to the documentation given by the respective design teams this is partly due to the fact that during the design, whenever complexity could be added ‘at no additional cost', it was added. ‘At no additional cost' should be understood as `no additional cost *on the Pentium Pro*'. On other ‘unknown' platforms [Cl99], these extra operations could be cheaply available, or not.

Serpent introduced asymmetry across the rounds by adopting 8 different S-boxes and asymmetry in the round transformation by having shift (instead of cyclic shift) operations. RC6 has a reasonably symmetrical design. However, it still mixes XOR and arithmetic addication operations and it uses 32-bit multiplication.

Another important advantage of Rijndael is that it was designed right from the start to support 128 bit block lengths. Twofish and RC6 on the other hand, are obviously upgrades from their 64-bit predecessors, respectively Blowfish and RC5, and this shows in the design.

4.1 Symmetry

There is only a single S-box, since until now, no advantage has been demonstrated for the use of different S-boxes (as in Serpent, Twofish and MARS). This S-box is applied in parallel to all state bytes. Similarly, the linear transformation and the round key addition treat all state bytes in the same way and have rotational symmetry. The round function is the same for the complete cipher execution (unlike Serpent and MARS) as the differences in the round keys are considered to introduce sufficient asymmetry. This gives Rijndael the following advantages:

- **Parallelism:** among the finalists, Rijndael is by far the best suited to be implemented on processors with a parallel architecture[Cl99], that is expected to be the architecture of the future (Merced, McKinley, …). Moreover, a dedicated hardware implementation in which the Rijndael round is fully hardwired can give very high speed thanks to its short critical path[DaRi98].

- **Compactness:** the single S-box and the simplicity of the linear transformation allow to code Rijndael in a small number of bytes, relevant on smart cards. Moreover, a minimal dedicated hardware implementation of Rijndael can be built by hardwiring a single S-box and a single 4-byte to 1-byte linear transform[DaRi98].
- **Absence of arithmetic operations:** the description of Rijndael does not make any (hidden) assumptions on the coding of integers as a sequence of bits. One of the advantages of this is that Rijndael is immune for so-called big endian/little endian confusion and conversion problems.

4.2 Orthogonality of the components

In Rijndael, the round function is composed of a number of components each with their own contribution: S-boxes for non-linearity, round key XORing for key dependence and asymmetry, byte transposition for inter-word diffusion and an MDS transform for intra-word inter-byte diffusion. This design feature allows to get more easily a view on the security of the algorithm.

We have provable lower bounds for linear and differential probabilities based on the interaction of these components. These proofs make use of only a few macroscopic properties of the components and leave a lot of freedom on how these properties are actually attained. The advantage of this modular approach is that components may be replaced without affecting these lower bounds as long as the macroscopic properties hold. For example, in the hypothetical case that an attack would be launched that makes use of some specific property of the current S-box, it could be replaced by another one without affecting the lower bounds.

For the other AES finalists, the interaction between the different components is intricate and much harder to analyse and the act of replacing a single component turns a lot of the analysis performed obsolete.

4.3 Confidence

As a consequence of its clarity of design and good performance results, Rijndael attracted by far the most attention from cryptanalysts outside the design team. Although the other finalists seem to have been analyzed quite thoroughly by their own designers, history has shown that `friendly' cryptanalysis is not as effective. A number of attacks on reduced versions has been published. We can conclude that Rijndael has a sufficient security margin, and do so with a high level of confidence.

5. Extensions

Rijndael is the only AES finalist that supports other block lengths than 128 bits, namely 192 bits and 256 bits. Moreover, extensions are defined for all combination of block lengths and key lengths between 128 and 256 bits in steps of 32 bits [DaRi98].

The added value of the longer block lenghts is that the cipher can be used as the compression function of a collision-resistant iterated hash function. Note that a length of 128 bits was considered to be insufficient for SHA-1.

6. Multiple algorithms

The technical reasons for having multiple algorithms for the AES would be the fact that a single algorithm cannot be efficiently and securely implemented on all target platforms, or to have a backup in case the primary algorithm has been broken.

If Rijndael is chosen as the AES, there is no need for an alternative algorithm for the first reason as Rijndael is very efficient on all target platforms. Of course, if MARS or RC6 would be chosen, smart card application developers will see their performance and RAM availability go down and will tend to stick to good old Triple-DES if no alternative AES is available.

In practice, “having a backup in case the primary algorithm is broken” is a very expensive and cumbersome undertaking. It implies coding, testing and integrating both the primary and the backup algorithms in all products and applications where this backup is really taken seriously. If Rijndael is chosen as AES, the “backup” could be a Rijndael version with the number of rounds doubled. In this respect it is worth while to consider the actual risk. For the current standard DES, the most practical attack to date is exhaustive key search, an attack that was already known before its publication. The more sophisticated attacks, such as linear and differential cryptanalysis are very interesting and have learnt us a lot on how to design ciphers, but are no threat in the real world. The design teams of the AES finalist algorithms know their literature and have all used the experience obtained from analysing DES, FEAL, IDEA, … to build their ciphers. Hence, although new attacks may always be found, we think it is unlikely that they will be a security threat in real-world applications, whatever choice is made among the finalists.

7. References

[Bi99] E. Biham, "A note on comparing the AES candidates", AES 2.

[BAK98] E. Biham et al., “Serpent, a proposal for the Advanced Encryption Standard”, AES 1.

[Cl99] C. Clapp, “Instruction-level parallelism in AES candidates”, AES 2.

[Co99] B. Schneier et al., “Performance comparison of the AES submissions”, AES 2.

[DaRi98] J. Daemen and V. Rijmen, “Rijndael”, AES 1. Updated version from `http://www.esat.kuleuven.ac.be/˜rijmen/rijndael`

[DaRi99] J. Daemen and V. Rijmen, “Resistance against implementation attacks: a comparative study of the AES proposals”, AES 2.

[Ke99] G. Keating, “Performance analysis of AES candidates on the 6805 CPU core”, AES 2. Updated version from `http://www.ozemail.com.au/~geoffk/aes-6805/`.

[IBM98] C. Burwick et al., “Mars - a candidate cipher for AES”, AES 1.

[RC98] R. Rivest et al., “The RC6™ block cipher”, AES 1.

[Co99] B. Schneier et al., “Twofish, a block encryption algorithm”, AES 1.

[Li00] H. Lipmaa, AES cipher performance cross-table, available at `http://home.cyber.ee/helger`

The Case for Serpent

Ross Anderson, Eli Biham and Lars Knudsen

24th March 2000

Summary

Serpent should be chosen because it is the most secure of the AES finalists. Not only does it have ample safety margin, but its simple structure enables us to be sure that none of the currently known attacks will work. It is also simple to check that an implementation is correct. Although Serpent is not as fast as the other finalists on the 200 MHz Pentium machine used for round 1 benchmarking, this disadvantage largely disappears when we consider the likely platforms and applications of the 21st century. In hardware, for example, Serpent has easily the best performance, while on IA64 it's second.

1 Security

The most important requirement is stated succinctly in the AES announcement [7]: *'The security provided by an algorithm is the most important factor in the evaluation.'*

From the day in September 1997 when we started designing Serpent, we asked ourselves what protection requirements we were trying to meet. We concluded that AES needed to last for a useful service lifetime plus a human lifetime after that. That means at least a century. So we like the AES motto of a *'crypto algorithm for the twenty-first century'*. Also, if Moore's Law runs out sometime this century, then the AES might never be replaced. So the selectors should consider how their choice will look in the twenty-second century and beyond.

1.1 Advances in mathematics

An algorithm may break if someone comes up with a powerful new theory. We do not believe that the history of cryptanalysis is over. Although we have no real idea what the next hundred (or five hundred) years of mathematics will bring, there are three things we can do to future-proof a design.

First, a block cipher should be simple and easy to analyse. The DES algorithm had such a complex description that until the late 1980's no-one appears to have tried seriously to attack it. When they did, differential [5] and then linear [9] attacks were found – both of which can now be explained to bright students in a single 50-minute lecture.

Second, a block cipher should have more rounds than are needed to block today's attacks. Improvements in cryptanalysis usually increase the number of rounds required.

Third, a block cipher should use only well understood primitives. S-boxes and SP-networks have been around for over a quarter of a century, so it is less likely that surprising new attacks will be found on them.

Serpent was designed with all these considerations firmly in mind.

1.2 Engineering issues

Moore's Law may be the most obvious interaction between crypto security and engineering. But assurance is at least as important. If Moore's law continues, then 128-bit keys will be vulnerable in about a century; but many systems fail right now because of design and implementation errors.

Complicated algorithms are hard to implement correctly, and it is harder still to prove implementations to be correct. Serpent's simple design makes verification easier. It is so simple that it can be optimised in high level languages such and C and Ada. So a developer can avoid many of the errors that creep into assembly language routines.

Many secure systems are also vulnerable because of poor random number generators, memory remanence or other engineering failures (e.g., [3]). These risks provide an even more compelling argument for 256-bit keys than either Moore's Law or quantum computers. It would be nice if implementation failures became less common over time, but experience suggests the contrary. As systems get more complex, there are more things to go wrong.

1.3 Public confidence

Ciphers can also be damaged through erosion of public confidence.

Recall the effect which the invention of differential and then linear cryptanalysis had on the standing of DES. Neither of these attacks is practical: there are no DES applications known to us where an opponent might get hold of 2^{40} texts. Indeed, a prudent designer would normally never use any key for a 64-bit block cipher to encipher more than 2^{32} texts. Yet despite the discoverers' strenuous efforts to keep the story straight, differential and linear attacks became translated in the public mind to 'DES has been broken'. It's imprudent to expect the public to distinguish between practical attacks and 'certificational' attacks – attacks which require infeasibly large amounts of data or effort.

We have often been asked why, given that Serpent is secure today with at most 16 rounds, we do not allow 16 rounds – at least for 128-bit keys. The answer is this. Having experienced what happened to DES, we are concerned that, in perhaps 50 years' time, advances in mathematics will lead to a certificational attack on 16-round Serpent. As the other AES finalists have no more margin of safety than 16-round Serpent, they run a similar risk. (That is why we believe that they should have more rounds, rather than Serpent having less.) We think such an attack on Serpent is unlikely. But 'unlikely' isn't enough; the AES algorithm should have the highest achievable level of design assurance.

So we believe that the Advanced Encryption Standard should be 32-round Serpent with 256-bit keys. If people want to use less than 256 bits, or less than

32 rounds, then they should do so only with good reason, and understand that the two issues are orthogonal. The threats against 128-bit 32-round Serpent and 16-round 256-bit Serpent are different.

2 Performance

Many superficial analyses of the AES finalists have concluded that Serpent is half the speed of the other candidates, because we used twice as many rounds as we needed to. This is not accurate.

The three most important aspects of performance are hardware complexity, software speed and memory cost. We have already discussed memory usage extensively in [2]; this is the critical parameter for embedded and smartcard applications. Serpent does extremely well here. We will spend the rest of this section discussing hardware and software.

First, Serpent is the best of the AES finalists in hardware – even with the full 32 rounds. An independent team produced implementations for the Xilinx XCV1000 FPGA of RC6, Rijndael, Serpent and Twofish[1]. Serpent was the only finalist for which a fully pipelined implementation could be fitted into a single chip. Serpent was also by far the fastest, achieving a throughput of 5.04 Gbit/sec, versus 2.40 Gbit/sec for RC6, 1.94 Gbit/sec for Rijndael and 1.71 Gbit/sec for Twofish [6]. An NSA study of ASIC costs predicts 8.03 Gbit/sec for Serpent versus 5.163 for Rijndael, 2.171 for RC6 and 1.445 for Twofish [12].

Second, several AES finalists are heavily optimised for encrypting very large files on the Pentium II. But in most applications, key agility matters more, and this isn't likely to change any time soon.

Gigabit networks already demand encryption of ATM cell streams. This often won't be done in the end systems, as people rely increasingly on boundary control devices such as firewalls or guards to create virtual private networks. This is likely to mean changing the key every three blocks.

In low cost embedded systems, key changes are already common. In [2] we described a typical fielded electronic purse system where each transaction involved ten key set-ups and fourteen block cipher operations.

So we believe that most real applications will have one key change every 1–5 encryptions, and suggest for simplicity's sake that the benchmark should be the one natural in ATM networks, namely the average cost of one key change plus three block cipher operations. On this benchmark, Serpent does not badly across a wide range of platforms, especially the IA-64 architecture which will almost certainly be the standard for the next generation of PCs. According to engineers from Hewlett Packard, the relevant figures are [13]:

	MARS	RC6	Rijndael	Serpent	Twofish	Serpent is:
IA64	2965	3051	504	**2269**	2991	**2nd**
PA-RISC	3409	2686	666	**2415**	3453	**2nd**

[1] Although this team did not implement MARS, there seems no reason to suppose that MARS would do any better than RC6

The above figures are the average clock cycle costs, over encryption and decryption, of one key setup plus three block cipher operations. Even on Pentium, using this benchmark, Serpent is the third fastest algorithm when one combines the published cycle count figures from Gladman [8] and Osvik [11], and fourth fastest combining Worley et al [13] and Osvik. It's second and third respectively with Osvik's latest figures (2531 cycles on a K7). We hope to have stable and comparable figures by the May 15th deadline. NIST's results also show Serpent doing well on Ultrasparc II [4] (though unfortunately without clock cycle counts).

One of the main things to emerge from the extensive testing of round 2 finalists is that some algorithms achieve high throughput at the cost of slow key setup, while others are reasonably key agile. We believe that very many application designers will prefer the latter.

Another point is that some algorithms achieve high software throughput at the cost of high hardware complexity. We believe that the AES should have a simple hardware implementation.

We are not trying to claim that Serpent is the fastest algorithm. Speed was not the primary goal of the AES competition, and we designed Serpent according to the specification from NIST. What we do say is that Serpent's security was not bought at an unacceptable price in speed.

3 Miscellaneous

Much has been written recently about power analysis. One of us is currently doing an implementation of all five finalists on an 8051-based smartcard with no specific power analysis defences. As the bulk of the work is being done by students, full results aren't expected until the end of the academic year. But from what's known so far, we don't expect that any one finalist will be much superior to any other: just that the attack techniques will differ.

The likely solution to power analysis is hardware engineering, and a strong contender is dual-rail logic in which the current drawn is independent of the data. One of us is involved in such a project [10]. Dual-rail design is easier where one only has to worry about the simple logical operations used in Serpent, rather than operations with carry, and especially multiplications. So choosing Serpent as the AES will make the smartcard designer's job easier.

Finally, the claim that Serpent's whole key schedule has to be worked out in advance for decryption is incorrect. It is not necessary to apply the S-boxes during the forward computation.

4 Conclusion

Serpent should be chosen as the Advanced Encryption Standard. It's the fastest algorithm in hardware, and the second fastest in software on the IA-64 architecture. Above all, Serpent should be chosen because it's the most secure of the candidates.

References

1. RJ Anderson, E Biham, LR Knudsen, “Serpent: A Proposal for the Advanced Encryption Standard’, submitted to NIST as an AES candidate. A short version of the paper appeared at the AES conference, August 1998; both papers are available at `http://www.cl.cam.ac.uk/~rja14/serpent.html`
2. RJ Anderson, E Biham, LR Knudsen, “Serpent and Smartcards” in *Cardis 98*, Springer Verlag (2000) pp 257–264; also available at `http://www.cl.cam.ac.uk/~rja14/serpent.html`
3. RJ Anderson, MG Kuhn, “Low Cost Attacks on Tamper Resistant Devices” in *Security Protocols – Proceedings of the 5th International Workshop* (1997) Springer LNCS vol 1361 pp 125–136
4. LE Bassham III, “Efficiency Testing of ANSI C implementations of Round 2 Candidate Algorithms for the Advanced Encryption Standard”, to appear in the proceedings of the 3rd AES Candidate Conference
5. E Biham, A Shamir, *‘Differential Cryptanalysis of the Data Encryption Standard’* (Springer 1993)
6. AJ Elbirt, W Yip, B Chetwynd, C Paar, “An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists”, to appear in the proceedings of the 3rd AES Candidate Conference
7. “Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)”, in *Federal Register* September 12, 1997 (Volume 62, Number 177), pp 48051–48058
8. B Gladman, “Implementation Experience with AES Candidate Algorithms”, in *Proceedings of the 2nd AES Candidate Conference* (NIST, 1999) pp 7–14
9. M Matsui, “Linear Cryptanalysis Method for DES Cipher”, in *Advances in Cryptology — Eurocrypt 93*, Springer LNCS v 765 pp 386–397
10. SW Moore, RJ Anderson, MG Kuhn, “Improving Smartcard Security using Self-timed Circuit Technology”, Fourth ACiD-WG Workshop, Grenoble, ISBN 2-913329-44-6, 2000
11. DA Osvik, “Speeding Up Serpent”, to appear in the proceedings of the 3rd AES Candidate Conference
12. B Weeks, M Bean, T Rozylowicz, C Ficke, “Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms”, to appear in the proceedings of the 3rd AES Candidate Conference
13. J Worley, B Worley, T Christian, C Worley, “AES Finalists on PA-RISC and IA64: Implementations and Performance”, to appear in the proceedings of the 3rd AES Candidate Conference

Comments on Twofish as an AES Candidate

Bruce Schneier[*] John Kelsey[†] Doug Whiting[‡] David Wagner[§] Niels Ferguson[¶]

March 24, 2000

1 Introduction

In 1996, the National Institute of Standards and Technology initiated a program to choose an Advanced Encryption Standard (AES) to replace DES. Four years later, NIST is about to choose that standard. We, the authors of the Twofish algorithm, would like to express our continued support for Twofish.

2 Twofish

Twofish is our submission to the AES process. Since first proposing the algorithm in 1998, we have continued to perform extensive analysis of the cipher: both cryptanalysis and performance analysis. We believe that Twofish is the best AES candidate of the five finalist algorithms.

Security: Twofish was designed primarily with security in mind. To date the Twofish round function has proven to be the strongest round function of any of the finalists, with the best known attack being on 6 rounds of Twofish compared to at least 9 rounds for any of the other finalists.

Performance: Twofish is routinely one of the fastest AES candidates; it was designed to have good performance on a variety of hardware and software platforms, instead of being optimized for a single platform. Although Twofish is not the easiest algorithm to implement or optimise, it is amongst the fastest algorithms on virtually every platform when properly implemented.

Flexibility: Twofish is unique in its implementation flexibility. The algorithm can be optimized for bulk encryption, key agility, low gate count, high gate count, or any combination of factors. All of these implementations are completely interoperable.

More interesting than these individual measures is the security/performance ratio of Twofish. Looking at the five algorithms in this manner—normalizing to the largest number of rounds cryptanalyzed is a good metric—Twofish far surpasses the other four finalists.

3 Discussion

The AES process has worked even better than expected. Today we have five good algorithms, and any of the designs would make a good AES standard. (We would recommend increasing the number of rounds for RC6 from 20 to 32, and the number of rounds in Rijndael from 10/12/14 to 18, to get at least a 2x security

[*] Counterpane Internet Security, Inc., 3031 Tisch Way, 100 Plaza East, San Jose, CA 95128, USA; schneier@counterpane.com.
[†] Counterpane Internet Security, Inc. kelsey@counterpane.com.
[‡] Hi/fn, Inc., 5973 Avenida Encinas Suite 110, Carlsbad, CA 92008, USA; dwhiting@hifn.com.
[§] University of California Berkeley, Soda Hall, Berkeley, CA 94720, USA; daw@cs.berkeley.edu.
[¶] Counterpane Internet Security, Inc. niels@counterpane.com.

margin—number of rounds greater than the maximum number of rounds that can be cryptanalyzed—as recommended by Lars Knudsen.)

Two of the finalists, MARS and RC6, are not well-suited certain applications, most notably small-memory implementations (e.g., smart cards) and highly key-agile systems (e.g., IPsec). Any one of the other three algorithms—Rijndael (with the extra rounds), Serpent, or Twofish would make an *excellent* standard.

Of the five finalists, Twofish has the best speed/security-margin tradeoff, as well as the most flexibility. With security and speed being the most important criteria (certainly the most talked-about), we believe that Twofish is the best single finalist.

4 More Information

More information on Twofish can be found on the Twofish Web site, at `http://www.counterpane.com/twofish.html`.

NOTES