A Comprehensive Framework for Fair and Efficient Benchmarking of Hardware Implementations of Lightweight Cryptography*

Jens-Peter Kaps[1], William Diehl[2], Michael Tempelmeier[3], Farnoud Farahmand[1], Ekawat Homsirikamol[4] and Kris Gaj[1]

[1] Cryptographic Engineering Research Group
George Mason University, Fairfax, Virginia 22030, USA
email: {jkaps, ffarahma, kgaj}@gmu.edu
[2] Signatures Analysis Lab
Virginia Tech, Blacksburg, Virginia 24061, USA
email: wdiehl@vt.edu
[3] Lehrstuhl für Sicherheit in der Informationstechnik
Technische Universität München, 80333 München, Germany
email: michael.tempelmeier@tum.de
[4] Independent Researcher
email: ekawat@gmail.com

Abstract. In this paper, we propose a comprehensive framework for fair and efficient benchmarking of hardware implementations of lightweight cryptography (LWC). Our framework is centered around the hardware API (Application Programming Interface) for the implementations of lightweight authenticated ciphers, hash functions, and cores combining both functionalities. The major parts of our API include the minimum compliance criteria, interface, and communication protocol supported by the LWC core. The proposed API is intended to meet the requirements of all candidates submitted to the NIST Lightweight Cryptography standardization process, as well as all CAESAR candidates and current authenticated cipher and hash function standards. In order to speed-up the development of hardware implementations compliant with this API, we are making available the LWC Development Package and the corresponding Implementer's Guide. Equipped with these resources, hardware designers can focus on implementing only a core functionality of a given algorithm. The development package facilitates the communication with external modules, full verification of the LWC core using simulation, and generation of optimized results. The proposed API for lightweight cryptography is a superset of the CAESAR Hardware API, endorsed by the organizers of the CAESAR competition, which was successfully used in the development of over 50 implementations of Round 2 and Round 3 CAESAR candidates. The primary extensions include support for optional hash functionality and the development of cores resistant against side-channel attacks. Similarly, the LWC Development Package is a superset of the part of the CAESAR Development Package responsible for support of Use Case 1 (lightweight) CAESAR candidates. The primary extensions include support for hash functionality, increasing the flexibility of the code shared among all candidates, as well as extended support for the detection of errors preventing the correct operation of cores during experimental testing. Overall, our framework supports (a) fair ranking of candidates in the NIST LWC standardization process from the point of view of their efficiency in hardware before and after the implementation of countermeasures against side-channel attacks, (b) ability to perform benchmarking within the limited time devoted to Round 2 and any subsequent rounds of the NIST LWC standardization process, (c) compatibility among implementations of the same algorithm by different designers and (d) fast deployment of the best algorithms in real-life applications.

Keywords: Lightweight cryptography · Authenticated cipher · Hash function · Hardware · API · Side-Channel Analysis

*This work is supported by the Department of Commerce (NIST) Grant no. 70NANB18H219

1 Introduction

Among the major cryptographic competitions, the first attempt at defining a hardware API (Application Programming Interface) took place during the SHA-3 contest [1], [2]. In the area of high-speed implementations, all 14 Round 2 candidates, all 5 Round 3 candidates, and the previous standard SHA-2 were implemented using the proposed interface and communication protocol by the group from George Mason University (GMU) [1]–[3]. This interface and protocol were then extended to the case of lightweight applications and applied to the implementations of 13 Round 2 and 5 Round 3 SHA-3 candidates [4]. Alternative interfaces of hash function cores were proposed in [5], [6]. No specific interface was endorsed by NIST as a requirement for all implementations.

During the subsequent CAESAR contest (Competition for Authenticated Encryption: Security, Applicability, and Robustness), conducted in the period 2013-2019, all major decisions were made by the CAESAR Committee, composed of 18 renowned cryptographers, representing multiple institutions worldwide [7].

The first version of the proposed hardware API for CAESAR was reported in [8]. This version was later substantially revised, endorsed by the CAESAR Committee in May 2016, and published as a Cryptology ePrint Archive in June 2016 [9]. A relatively minor addendum was proposed in the same month, and endorsed by the CAESAR Committee in November 2016 [10].

The commonly accepted CAESAR Hardware API provided the foundation for the GMU Development Package, released in May and June 2016 [11]. This package included in particular: a) VHDL code of a generic PreProcessor, PostProcessor, and CMD FIFO, common for all Round 2 and Round 3 CAESAR Candidates (except Keyak), as well as AES-GCM, b) Universal testbench common for all API-compliant designs (aead_tb), c) Python app used to automatically generate test vectors (aeadtvgen), and d) Reference implementations of several dummy authenticated ciphers.

This package was accompanied by the Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API, v1.0, published at the same time [12]. A few relatively minor weaknesses of this version of the package, discovered when performing experimental testing using general-purpose prototyping boards, were reported in [13], [14].

In December 2017, a substantially revised version of the Development Package (v.2.0) and the corresponding Implementer's Guide were published by the GMU Benchmarking Team [11], [15]. The main revisions included a) Support for the development of lightweight implementations of authenticated ciphers, b) Improved support for the development of high-speed implementations of authenticated ciphers, and c) Improved support for experimental testing using FPGA boards, in applications with intermittent availability of input sources and output destinations.

It should be stressed that at no point was the use of the Development Package required for compliance with the CAESAR Hardware API. To the contrary, [12] clearly stated that the implementations of authenticated ciphers compliant with the CAESAR Hardware API could also be developed without using any resources belonging to the package [11], by just following the specification [9] directly.

In spite of being non-mandatory and the lack of official endorsement by the CAESAR Committee, the CAESAR Development Package played a significant role in increasing the number of implementations developed during Round 2 of the CAESAR contest. Out of 43 implementations reported before the end of Round 2, 32 were fully compliant, and one partially compliant with the CAESAR Hardware API. All fully compliant code used the GMU Development Package. The fully and partially compliant implementations covered 28 out of 29 Round 2 candidates (all except Tiaoxin) [11]. In Round 3, the submission of the hardware description language code (VHDL or Verilog) was made obligatory by the CAESAR Committee. As a result, the total number of designs reached 27 for 15 Round 3 candidates. Out of these 27 designs, 23 were fully compliant and 1 partially compliant with the CAESAR Hardware API [11]. Overall, publishing the CAESAR Hardware API, as well as its endorsement by the organizers of the contest, had a major influence on the fairness and the comprehensive nature of the hardware benchmarking during the CAESAR competition.

Several optimized lightweight implementations compliant with the CAESAR API, and based on v.2.0 of the Development Package, were reported in [16]. In [17]–[20], several other implementations were enhanced with countermeasures against Differential Power Analysis. In order to facilitate this enhancement, an additional Random Data Input (RDI) port was added to the CAESAR Hardware API.

In this paper, we propose leveraging all resources developed and experiences gained during the CAESAR competition, and applying them toward the development of a comprehensive framework for fair and efficient

hardware benchmarking of candidates during Round 2 and any subsequent rounds of the NIST Lightweight Cryptography standardization process. In Section 2, we describe the proposed hardware API for lightweight cryptography, with a special focus on extensions compared to the CAESAR Hardware API [9], [10]. In Section 3, we summarize the features of the LWC Development Package, including all extensions compared to the corresponding parts of the CAESAR Development Package. In Section 4, we provide the suggested timeline for the comprehensive hardware benchmarking effort focused on candidates qualified to Round 2 of the NIST Lightweight Cryptography standardization process.

2 Hardware API for Lightweight Cryptography

2.1 Minimum Compliance Criteria

The main reasons for defining a common API for all hardware implementations of candidates submitted to the NIST Lightweight Cryptography standardization project [21] are: a) Fairness of benchmarking, b) Compatibility among implementations of the same algorithm by different designers, and c) Ease of creating the supporting development package, aimed at simplifying and speeding up the design process. The recommended minimum compliance criteria, supporting all the aforementioned objectives, are listed below.

Authenticated encryption and decryption should be implemented within one core, but only one of these two operations should be executed at a time (half-duplex). If a given algorithm supports hashing, then designers should develop two versions of the LWC core, capable of performing a) encryption, decryption, and hashing, and b) encryption and decryption only.

Key scheduling of authenticated ciphers should be fully implemented within the LWC core. The LWC core should properly handle incomplete blocks in associated data (AD), plaintext, hash message, and ciphertext. In particular, padding should be implemented in hardware, and any unused portions of the last block released to the output should be cleared (filled with zeros).

The decrypted plaintext blocks should be released immediately, without waiting for the result of authentication. We assume that the delayed release of decrypted data, dependent on the result of authentication, will be handled by an external circuit, which is FIFO-based and similar for each candidate. Storing a decrypted plaintext internally, until the result of the verification is known would a) complicate the design and benchmarking, and b) make the calculation of the output latency and throughput dependent on the output buffer size and implementation details (e.g., support for simultaneous reading and writing).

The LWC core should support only inputs composed of full bytes. The core should support empty AD, plaintext, hash message, ciphertext, and any meaningful combination thereof. *For the purpose of benchmarking, the LWC core should support at least the following maximum sizes of associated data, plaintext, and hash messages:*

For single-pass algorithms:

Sa) $2^{16}-1$ *: default; used for comparison with implementations of two-pass algorithms*

Sb) $2^{32}-1$ *: kept for compatibility with the CAESAR API; practical only for single-pass algorithms*

Sc) $2^{50}-1$ *: minimum limit established by NIST for algorithms submitted to the Lightweight Cryptography standardization process.*

For two-pass algorithms:

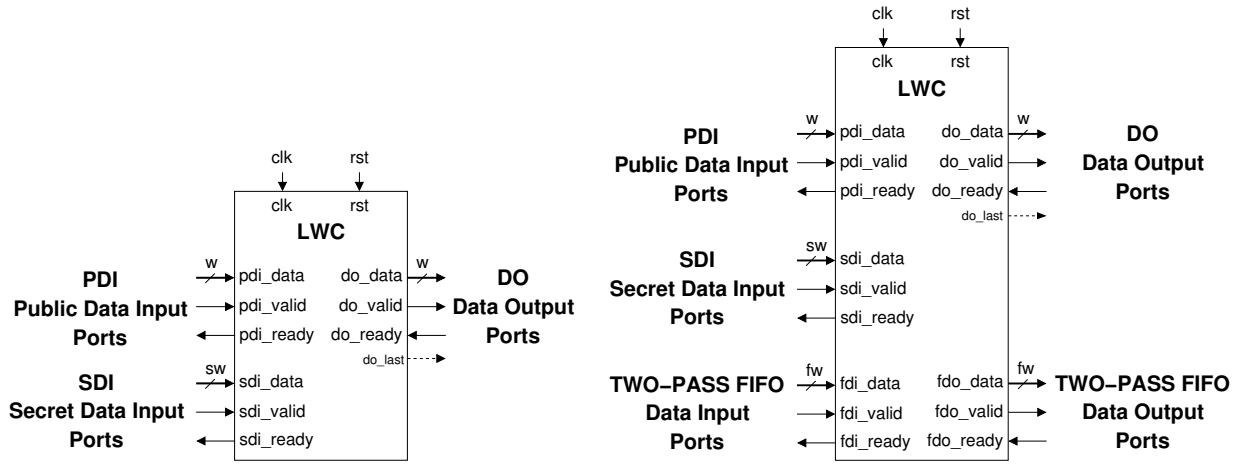
Ta) $2^{16}-1$ *: default; used for comparison with implementations of single-pass algorithms*

Tb) $2^{11}-1$ *: kept for compatibility with the CAESAR API*

Tc) $2^{50}-1$ *: minimum limit established by NIST for algorithms submitted to the Lightweight Cryptography standardization process.*

The core should also support the corresponding ciphertext sizes. However, the size limit $2^{16}-1$ *should be treated as default, and the implementers should do their best to eliminate (or at least minimize to negligible) the influence of the remaining size limits on the a) maximum clock frequency, b) total number of clock cycles for short messages, c) throughput for long messages.*

The use of external memory is allowed only for two-pass algorithms, and only for results of the first pass. For single-pass algorithms, no external memory should be used. Two-pass algorithms can typically benefit from external memory, used to store intermediate values utilized as inputs to the second pass. An alternative is to provide an entire input for the second time to the data inputs of the LWC core. However, doing that is



(a) LWC interface for single-pass algorithms

(b) LWC interface for two-pass algorithms

Figure 1: LWC interface options

typically less efficient in terms of throughput. Additionally, providing the same input twice through the same port complicates the input circuit, e.g., by requiring two costly DMA transfers, or placing external memory and the associated control logic before the data input ports.

The core should have only one clock input and one internal clock signal. The implementation should be able to operate at the maximum clock frequency determined by the critical path located entirely inside of the LWC core. Using a single clock domain simplifies static timing analysis, generation of post-place and route results, and optimization of FPGA tool options.

Parts of the data inputs that are not changed by encryption or decryption operations, respectively, should not be passed to the output. In particular, Npub and AD should not be a part of the output from either encryption or decryption (see Fig. 4). This assumption removes the need for any bypass FIFO necessary to pass any unchanged data to the output. Any formatting of output from encryption/decryption, for the purpose of transmission through the network or subsequent decryption/encryption, respectively, is assumed to be performed outside of the LWC core.

The permitted widths of the data buses for the Public Data Input (PDI), Secret Data Input (SDI), and Data Output (DO) ports are as follows: PDI and DO: $w = 8$, 16, or 32; SDI: $sw = 8$, 16, or 32. 8-bit, 32-bit, and 16-bit processors are among the most popular processors used in embedded systems, especially in resource constrained environments. An LWC core needs to be able to communicate with at least one of these processors. Hardware architectures of lightweight ciphers and hash functions often use the internal datapath width equal to 8, 16, or 32 bits. It is quite natural (although not required) for an external data bus width to match the internal datapath width. The permitted widths of external data buses also match those defined in the CAESAR Hardware API [9]. This feature makes all existing lightweight implementations of CAESAR candidates compatible with the proposed hardware API, which provides many helpful reference points for the comparison of results, as well as many helpful open-source examples.

2.2 Interface

The proposed interface of the LWC core for single-pass algorithms is shown in Fig. 1a. This interface includes three major data buses: `PDI`, `SDI`, and `DO`, as well as the corresponding handshaking control signals, named *valid* and *ready*. The *valid* signal indicates that the data is ready at the source, and the *ready* signal indicates that the destination is ready to receive it. The signal *do_last* is an optional signal which simplifies the connection to an *AXI4-Stream Slave* (see Fig. 2). The physical separation of Public Data Inputs (such as the plaintext, AD, public message number, etc.) from Secret Data Inputs (such as the key) is dictated by the resistance against any potential attacks aimed at accepting public data, manipulated by an adversary, as a new key.

The handshaking signals are a subset of major signals used in the AXI4-Stream interface [22]. As a result,

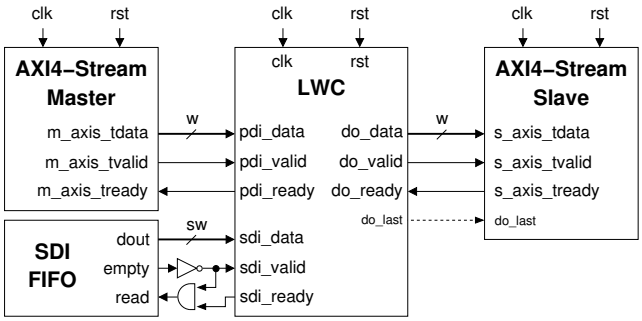


Figure 2: Typical external circuits: AXI4-Stream IPs

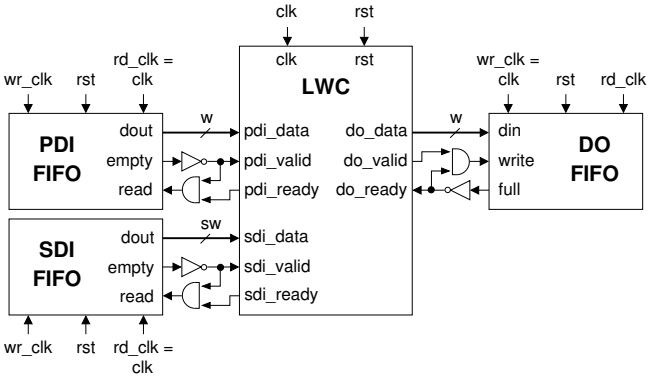


Figure 3: Typical external circuits: FIFOs

LWC can communicate directly with the *AXI4-Stream Master* through the Public Data Input, and with the *AXI4-Stream Slave* through the Data Output, as shown in Fig. 2. At the same time, *LWC* is also capable of communicating with much simpler external circuits, such as FIFOs, as shown in Fig. 3. An advantage of using FIFOs at all data ports is their potential role as suitable boundaries between two clock domains, one used for communication and one for computation. This role is facilitated by the use of separate read and write clocks, shown in Fig. 3 as `rd_clk` and `wr_clk`, respectively.

The reset input can be either synchronous or asynchronous, and either active-high or active-low, depending on the conventions used in a given technology (e.g., FPGA vs. ASIC), as well as the personal preference of the designers.

The recommended interface of two-pass algorithms is shown in Fig. 1b. Compared to the interface of single-pass algorithms, shown in Fig. 1a, additional ports used for communication with the external Two-Pass FIFO have been added. The width of the data buses of these ports is defined by a constant, denoted in Fig. 1b as fw. The value of this constant can be selected freely by the designers, depending on the specific feature of each two-pass algorithm and its implementation. In modern FPGAs, the Two-Pass FIFO will be implemented using block memories (such as BRAMs of Xilinx FPGAs and embedded memory blocks of Intel FPGAs). A FIFO with a capacity of 2^{16} bytes can be built using a relatively small percentage of the total size of on-chip block memories. Thus, the two-pass algorithms are not in any significant way disadvantaged compared to single-pass algorithms. Additionally, even for single-pass algorithms, a NIST compliant implementation of authenticated decryption is expected to store the deciphered plaintexts until the authenticity of the plaintext is verified. Only then, the plaintext is allowed to be released. Implementing this feature in hardware would require the memory approximately equal in size to the two-pass FIFO. Taking these considerations into account, the two-pass FIFO is treated as an external circuit, located outside of the LWC core, and, as such, should not affect either the resource utilization or the maximum clock frequency of the LWC core.

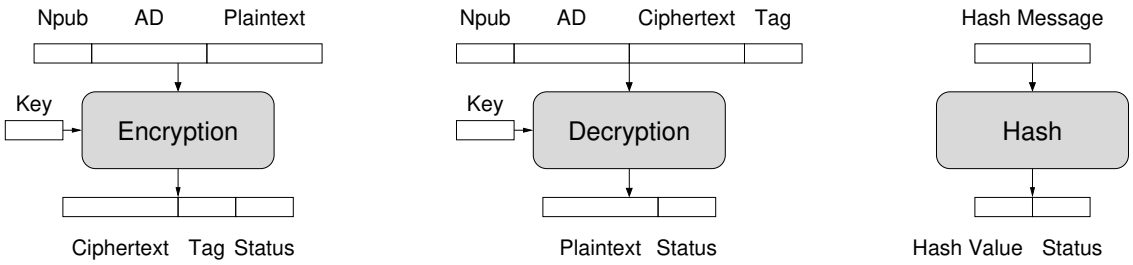


Figure 4: Input and Output of an Authenticated Cipher and a Hash Function. Notation: Npub - Public Message Number, AD - Associated Data

instruction = LDKEY
seg_0_header
seg_0 = Key

Figure 5: Format of Secret Data Input for loading the key

2.3 Communication Protocol

All parts of a typical input and a typical output of an authenticated cipher and a hash function are shown in Fig. 4. The proposed format of the Secret Data Input is shown in Fig. 5. The entire input starts with an instruction, which in the case of `SDI` is limited to Load Key (LDKEY). The instruction is followed by segments. Each segment begins with a separate header, describing its type and size. In the case of `SDI`, the only segment type necessary to meet the minimum compliance criteria is: `Key`, denoting a string of bits carrying an authenticated cipher key.

The proposed format of the Public Data Input is shown in Figs. 6(a)(c) and 7(a)(c). The allowed instruction types are Activate Key (ACTKEY), Authenticated Encryption (ENC), Authenticated Decryption (DEC), and Hash. The Activate Key instruction, typically directly precedes the Authenticated Encryption or Authenticated Decryption instruction. `Public Data Input (PDI)` is divided into segments. Each segment starts from the segment header, describing its type and length. If no data is to be sent to the LWC core for a segment, in case of AD, Plaintext, Ciphertext, and Hash Message segments, the segment header still has to be sent with the Segment Length field of the respective header set to 0.

The AD, Plaintext, and Hash Message can be (but do not have to be) divided into multiple segments (as shown in Fig. 7(c)). *The maximum size of each segment is assumed to be* $2^{16} - 1$ *bytes.* The primary reason for dividing AD, Plaintext, and Hash Message into multiple segments is that the full input size may be unknown when the authenticated encryption or hashing starts. Npub can only use one segment, as its size is typically quite small (in the range of 16 bytes).

Figures 6 and 7 present the typical format of input (`PDI`) and output (`DO`) of authenticated encryption, decryption, and hash operation, respectively, for the ciphers that do not use Nsec. The order of segment types that can be processed by a given core is a feature of the implemented algorithm and needs to be clearly documented.

2.4 Side-channel Resistant Implementations

The NIST LWC Standardization Process does not mandate, but does encourage, algorithms and implementations that support effective and efficient side-channel countermeasures [21]. This includes implementations secure against power analysis side-channel attacks (SCA), such as Simple Power Analysis (SPA) and Differential Power Analysis (DPA).

One common requirement for nearly all algorithmic power analysis countermeasures, e.g., Boolean Masking and Threshold Implementations (TI), is the consumption of randomness during cipher operations. An example is the necessity of meeting the TI uniformity property, which often requires so-called refreshing randomness [23], [24]. To facilitate side-channel resistant implementations that require refreshing randomness, we propose an additional `Random Data Input (RDI)` bus, comprised of the signals `rdi_data` (of user selectable width

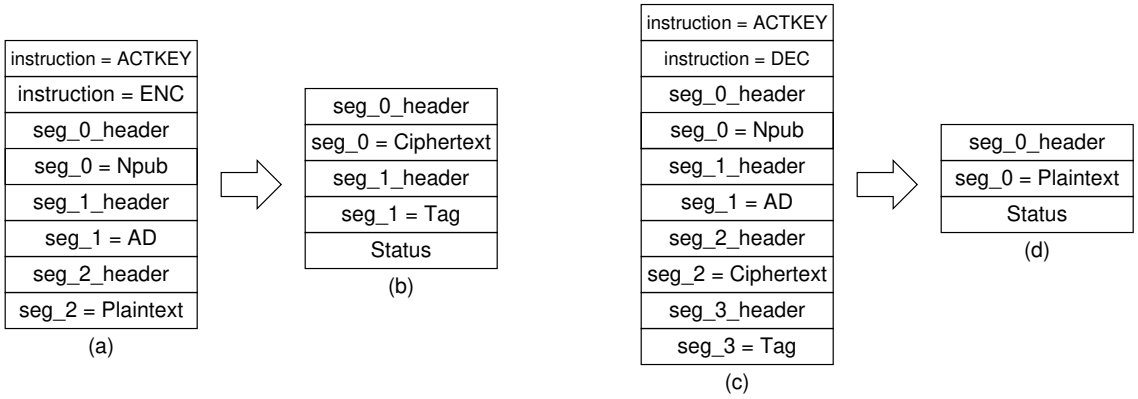


Figure 6: Format of Public Data Input (PDI) and Data Output (DO) for authenticated encryption a) PDI, b) DO and authenticated decryption c) PDI, d) DO

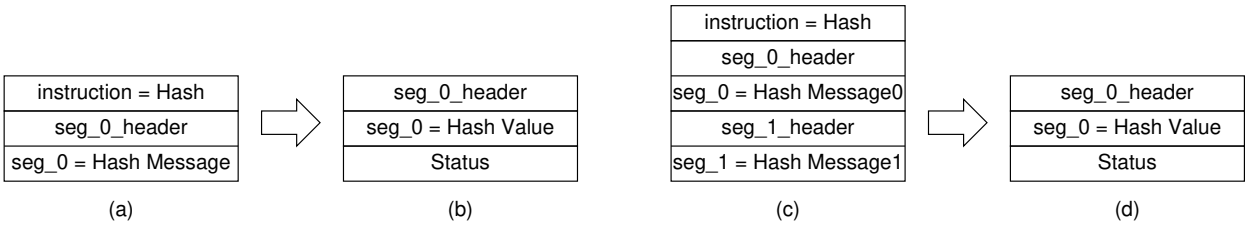


Figure 7: Format of Public Data Input (PDI) and Data Output (DO) for hashing with one segment for the Hash Message a) PDI, b) DO and with multiple segments for the Hash Message c) PDI, d) DO

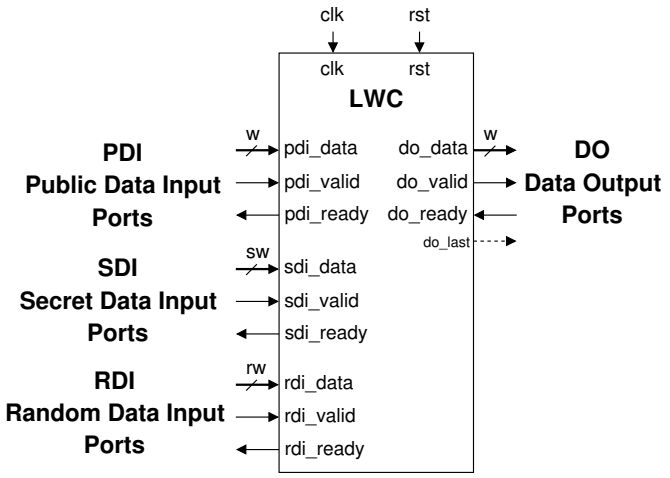
rw), `rdi_ready` and `rdi_valid` (see Fig. 8a). No protocol support is provided for this optional interface. The protected LWC core simply asserts the `rdi_ready` signal, checks `rdi_valid` and then reads rw bits of random data.

The LWC API makes no assumptions on which kind of SCA protection is being implemented. However, if data has to be separated into shares, this task should be performed in software. All shares should be sent to the LWC cipher through the regular inputs, i.e., all public data should still arrive via `PDI`, and depart via `DO`. For an n-share implementation, the Plaintext segments should then contain shares 1 through n, w bits each, followed by the next w bits of each share, and so on. In the same fashion, separated data should leave the core. Fig. 8b shows how a $m \cdot w/8$-byte Plaintext, split into n shares, should arrive at `PDI`.

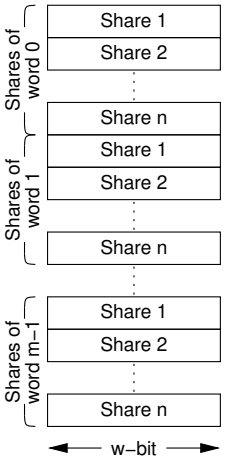
Share separation in software facilitates verification of countermeasures in hardware using leakage detection techniques (e.g., t-test or χ^2-test), for which false-positive results could occur if share separation were performed in hardware. The implementation designer should provide two scripts, one that takes as input the test vectors of an unprotected implementation and outputs the share separated data, and one that combines share separated outputs back, so that they can be compared to the test vectors for the output of an unprotected implementation.

2.5 Differences Compared to the CAESAR Hardware API

Major differences between the proposed Lightweight Cryptography Hardware API and the CAESAR Hardware API, defined in [9], [10], are as follows:
In terms of the Minimum Compliance Criteria: a) One additional configuration, encryption/decryption/hashing, has been added on top of the previously supported configuration: encryption/decryption. b) On top of the maximum sizes of AD/plaintext/ciphertext already supported in the CAESAR Hardware API, two additional maximum sizes, $2^{16}-1$ and $2^{50}-1$, have been added.
In terms of the Interface: An additional optional output, `do_last`, has been added to the Data Output ports.
In terms of the Communication Protocol: a) In the Instruction/Status, an additional opcode value, repre-



(a) LWC interface for the single-pass algorithms with random data input



(b) Plaintext/Ciphertext of $m \cdot w/8$ bytes split into n shares

Figure 8: Support for side-channel resistant implementations

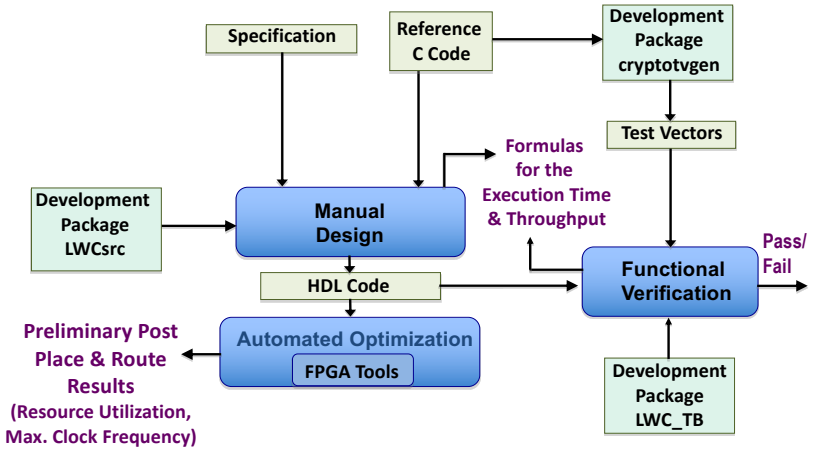


Figure 9: The API-Compliant Code Development using the Development Package

senting a hash function, has been added. b) In the Segment Header word, two additional Segment Type values, representing Hash Message and Hash Value, have been added.
In terms of Support for Side-Channel Resistant Implementations: No support for side-channel resistant hardware implementations was provided in the CAESAR Hardware API. This specification addresses this issue in Section 2.4, by defining a) An extended interface, shown in Fig. 8a, b) The requirement for the generation and merging of shares outside of the LWC core, and c) The mechanism for passing the input shares to the core and the output shares from the core, as shown in Fig. 8b.

3 Development Package and Implementer's Guide

To make our framework more efficient in terms of the hardware development time, the designers are provided with the following important resources, compliant with the use of the proposed LWC Hardware API:
a) VHDL code supporting the API protocol, common to all Lightweight Cryptography standardization process candidates, as well as all CAESAR candidates and AES-GCM (LWCsrc)
b) Universal testbench, common for all API-compliant designs (LWC_TB)
c) Python app used to automatically generate test vectors (cryptotvgen)

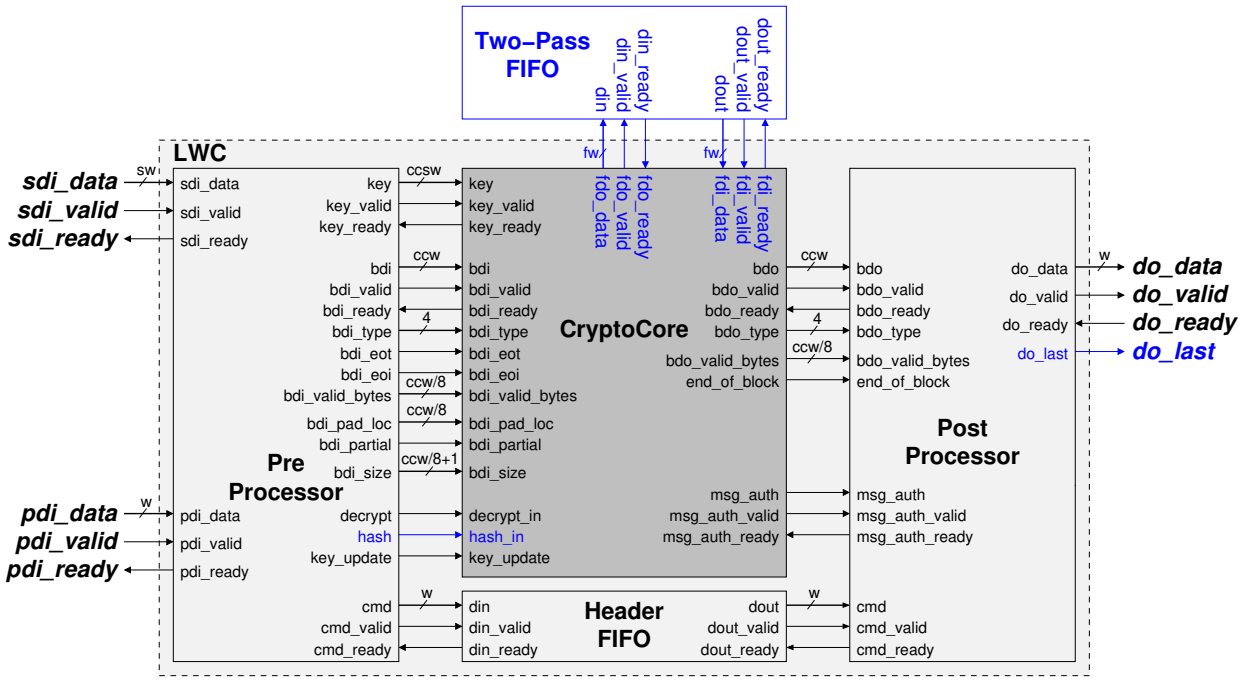


Figure 10: Top-level block diagram of the LWC core

d) Reference implementations of a dummy authenticated cipher and a dummy hash function
e) Implementer's Guide, describing all steps of the development and benchmarking process, including verification, experimental testing, and generation of results.

It should be stressed that the *implementations of authenticated ciphers (with an optional hash functionality), compliant with the LWC Hardware API, can also be developed without using any of the aforementioned resources, by just following the specification of the LWC Hardware API directly.* However, the necessary development time would be most-likely significantly longer, and the obtained results very comparable.

The major phases of the API-compliant code development process are summarized in Fig. 9. The manual design process is based on the specification and the reference C code of a given algorithm. The HDL code specific for a given algorithm is combined with the code shared among all algorithms, provided in the folder LWCsrc of the Development Package. Comprehensive test vectors are generated automatically by cryptotvgen based on the reference C code. These vectors are used together with the universal testbench, LWC_TB, to verify the HDL code using simulation. The verification is used to confirm the required functionality as well as formulas for the execution time and throughput derived during the design process. Automated optimization tools, such as Minerva [25], combined with FPGA tools, such as Vivado, are then used to generate post-place and route results, to be entered into the results database.

3.1 Block Diagram and Design Methodology

Fig. 10 shows the proposed top-level block diagram of the lightweight architecture of an authenticated cipher compliant with the LWC Hardware API. The top-level unit is made of four lower-level units called the *PreProcessor*, *CryptoCore*, *Header FIFO*, and *PostProcessor*.

The *PreProcessor* is responsible for the following tasks: a) parsing segment headers, b) loading keys, c) passing input blocks to the *CryptoCore*, along with information required for padding, and d) keeping track of the number of data bytes left to process. The *PostProcessor* is responsible for the following tasks: a) clearing any portions of output words not belonging to the ciphertext or plaintext, b) generating the header for the output data blocks, and c) generating the status block with the result of authentication. The *Header FIFO* is a small FIFO that temporarily stores all segment headers that need to be passed to the output.

Our Development Package supports the following dependencies between the data bus widths shown in Fig. 10: a) $sw = w$ (for w=8, 16, 32), b) Values of (w, ccw) pairs, where ccw is the `bdi` and `bdo` data bus

width, equal to (8, 8), (16, 16), (32, 32), as was the case in the CAESAR Development Package v.2.0, and (32, 8) and (32, 16), as two new variants. In general, when comparing the following variants: (32, 8) vs. (8, 8) and (32, 16) vs. (16, 16), we expect (a) negligible differences in throughput and latencies, especially for medium to long messages, and (b) minimal differences in area.

The code of the *PreProcessor*, *PostProcessor*, and *Header FIFO* is provided as a part of the Development Package. A designer only needs to develop the *CryptoCore*. The development of this module is left to individual designers and can be performed using their own preferred design methodology. An example design of the lightweight *CryptoCore* for a dummy authenticated cipher and a dummy hash function is provided as a part of our distribution. The corresponding code is developed to work correctly with $ccw = ccsw = 8$, 16, and 32. Multiple examples of full designs developed during the CAESAR competition are also available at [11].

3.2 Test Vector Generator (*cryptotvgen*) and Universal Testbench (*LWC_TB*)

The Python script called *cryptotvgen* and accompanying examples provide a framework to generate test vectors for any authenticated cipher, with optional hash functionality, based on the user's specified parameters. The framework relies on the reference implementations of the respective algorithms. The arguments of *cryptotvgen* are the function of the a) algorithm, b) parameters of the algorithm (e.g., key size, block size), and c) phase of verification. While it is possible to generate test vectors using pure shell command syntax, this process is likely to be error-prone due to a large number of available options. Instead, we recommend that the user creates a Python script that utilizes *cryptotvgen* as a third party library in Python, and then calls it using *cryptotvgen(args)*. Various examples of such Python scripts are provided as a part of the Development Package.

A typical process of verifying the functionality of an authenticated cipher module includes several phases, devoted to the verification of: a) a Single AD/Plaintext/Ciphertext/Hash Message block, b) Random inputs with custom selected sizes, c) Empty AD/Plaintext/Ciphertext/Hash Message, and d) randomly generated test vectors with varying AD, Plaintext, Ciphertext, and Hash Message lengths. The designer has the flexibility of generating his/her own verification strategy, based on the detailed knowledge and understanding of options of *cryptotvgen*. Test vectors should be selected in such a way that all corner cases specific to a given algorithm are covered. Particular attention should be paid to the generation of inputs with partial blocks.

Once test vectors are generated, they should be copied to the simulation folder. The simulation is performed using the universal testbench (*LWC_TB*) until the end-of-file is reached or a mismatch between expected output and actual output occurs. The testbench can also be configured to ignore errors and allow the simulation to run until the specified time.

In practical applications, there is no guarantee that the input source will be ready with the new data whenever the LWC core attempts to read it. Similarly, the destination circuit may not always be ready to receive a new output. These conditions must be comprehensively verified using simulation before the experimental testing is attempted. The provided testbench can be configured with specific values of the intervals at which the source or destination are not ready to exchange data with the LWC core, expressed in clock cycles.

3.3 Generation and Publication of Results

The generation of results is possible for the *LWC* core and the *CryptoCore*. We recommend generating results primarily for the *LWC* cores. Benchmarking and reporting results for FPGAs should be performed using the most-recent low-cost families of FPGA devices from at least two major vendors, Intel and Xilinx. For Intel, such families include Cyclone V and Cyclone 10 FPGAs, and Cyclone V SoC FPGAs; for Xilinx Artix-7 and Spartan-7 FPGAs, and Zynq-7000 All Programmable SoCs. The most recent versions of tools from the respective vendors should be used. Only the final results obtained after placing and routing should be reported. In terms of optimization of tool options, for Xilinx FPGAs and SoCs, we recommend generating results using Minerva [25]. In the case of ASICs, state-of-the-art libraries of standard cells should be used. Comprehensive results, generated after the respective submission deadlines for the hardware description language code, are expected to be made publicly available in the ATHENa Database of Results for Authenticated Ciphers [26] or an equivalent or extended database of results, focused on LWC candidates.

3.4 Experimental Testing

The framework from [13] and its extended version from [14] can be used for hardware testing. This framework is based on the use of the Xilinx PYNQ board, including the Zynq-7000 All Programmable SoC. This open-source project provides a hardware testbed for authenticated ciphers. It uses the Processing System (PS) of Zynq SoC to generate test vectors. It then sends these test vectors to the Programmable Logic (PL) and reads the results back, with the use of the Xilinx Direct Memory Access (DMA) to AXI4-Stream (AXIS) controllers. Additionally, it features two hardware timers to measure the time needed in the core itself and the overhead required to send data to and from the authenticated cipher core through DMA. It supports on-chip power measurements and determining the maximum clock frequency using experimental testing.

This framework has been used successfully to locate errors in the HDL code of CAESAR candidates [13], [14], preventing the corresponding implementations from running properly on the board. Even though the generation of primary timing and resource utilization results does not require experimental testing, the detected errors and the follow-up changes in the code may influence the final results. Additionally, experimental measurements of power consumption and maximum clock frequency can be used to verify the accuracy of the respective FPGA tools and the validity of assumptions used by these tools.

4 Proposed Timeline and Future Work

The initial version of the Hardware API for Lightweight Cryptography was submitted for discussion at the lwc-forum on July 17, 2019. The received comments were addressed through direct responses sent to the lwc-forum, as well as changes incorporated into the LWC Hardware API, Development Package, and Implementer's Guide. All the aforementioned resources were made publicly available on October 14, 2019.

For the maximum effectiveness of the hardware benchmarking process, we highly recommend that NIST endorses the final version of the proposed API, reflecting the consensus of the cryptographic engineering community. The authors also suggest that NIST enforces the submission of the hardware description language code, compliant with this Hardware API, for all candidates qualified to Round 2. The deadline for such submissions could be set in the middle of Round 2, e.g., to January 31, 2020. This first deadline should concern only implementations unprotected against side-channel attacks, supported by the Development Package released on October 14, 2019.

The GMU team is planning to a) support all Round 2 submission teams with their hardware implementation efforts, by providing technical support regarding the aforementioned Development Package and its documentation, b) take responsibility for the uniform implementation of a significant subset of all Round 2 candidates. After the deadline for submitting unprotected hardware implementations passes, e.g., in February 2020, our team would be happy to perform the comprehensive benchmarking of all submitted codes, publish the obtained results in an online database, and develop the corresponding written report with the thorough analysis of the obtained results. Shortly after, our team will also review any comments received from NIST and the entire cryptographic community and release the revised and extended version of the Development Package and the corresponding Implementer's Guide, supporting the design of implementations protected against side-channel attacks. The deadline for submitting such implementations could be then set to the date coinciding with the deadline for the complete Round 3 submissions.

References

[1] K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. LNCS, vol. 6225, Santa Barbara, CA, Aug. 2010, pp. 264–278.

[2] E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," Cryptology ePrint Archive 2010/445, 2010.

[3] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, "Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs," Cryptology ePrint Archive 2012/368, 2012.

[4] J.-P. Kaps, K. K. Surapathi, B. Habib, S. Vadlamudi, S. Gurung, and J. Pham, "Lightweight Implementations of SHA-3 Candidates on FPGAs," in *12th International Conference on Cryptology in India, Indocrypt 2011*, ser. LNCS, vol. 7107, Chennai, India, Dec. 2011, pp. 270–289.

[5] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane, "FPGA Implementations of the Round Two SHA-3 Candidates," in *2010 International Conference on Field Programmable Logic and Applications, FPL 2010*, Milan, Italy, Aug. 2010, pp. 400–407.

[6] M. Knezevic, K. Kobayashi, J. Ikegami, S. Matsuo, A. Satoh, Ü. Kocabas, J. Fan, T. Katashita, T. Sugawara, K. Sakiyama, I. Verbauwhede, K. Ohta, N. Homma, and T. Aoki, "Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA-3 Candidates," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 5, pp. 827–840, May 2012.

[7] *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness - web page*, https://competitions.cr.yp.to/caesar.html, 2019.

[8] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, M. U. Sharif, and K. Gaj, "A universal hardware API for authenticated ciphers," in *2015 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015*, Riviera Maya, Mexico, Dec. 2015.

[9] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, P. Yalla, J.-P. Kaps, and K. Gaj, "CAESAR Hardware API," Cryptology ePrint Archive 2016/626, 2016.

[10] ——, "Addendum to the CAESAR Hardware API v1.0," George Mason University, Fairfax, VA, GMU Report, Jun. 2016.

[11] Cryptographic Engineering Research Group (CERG) at George Mason University, *Hardware Benchmarking of CAESAR Candidates*, https://cryptography.gmu.edu/athena/index.php?id=CAESAR, 2019.

[12] E. Homsirikamol, P. Yalla, F. Farahmand, W. Diehl, A. Ferozpuri, J.-P. Kaps, and K. Gaj, "Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API," George Mason University, Fairfax, VA, GMU Report, 2016.

[13] M. Tempelmeier, F. De Santis, G. Sigl, and J.-P. Kaps, "The CAESAR-API in the real world — Towards a fair evaluation of hardware CAESAR candidates," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2018*, Washington, DC, Apr. 2018, pp. 73–80.

[14] M. Tempelmeier, G. Sigl, and J.-P. Kaps, "Experimental Power and Performance Evaluation of CAESAR Hardware Finalists," in *2018 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2018*, Cancun, Mexico, Dec. 2018, pp. 1–6.

[15] P. Yalla and J.-P. Kaps, "Evaluation of the CAESAR hardware API for lightweight implementations," in *2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017*, Cancun, Mexico, Dec. 2017.

[16] F. Farahmand, W. Diehl, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "Improved Lightweight Implementations of CAESAR Authenticated Ciphers," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018*, Boulder, CO, Apr. 2018, pp. 29–36.

[17] W. Diehl, A. Abdulgadir, F. Farahmand, J.-P. Kaps, and K. Gaj, "Comparison of cost of protection against differential power analysis of selected authenticated ciphers," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2018*, Washington, DC, Apr. 2018, pp. 147–152.

[18] ——, "Comparison of Cost of Protection against Differential Power Analysis of Selected Authenticated Ciphers," *Cryptography*, vol. 2, no. 3, p. 26, Sep. 2018.

[19] W. Diehl, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "Face-off between the CAESAR Lightweight Finalists: ACORN vs. Ascon," in *2018 International Conference on Field Programmable Technology, FPT 2018*, Naha, Okinawa, Japan, Dec. 2018.

[20] ——, "Face-off between the CAESAR Lightweight Finalists: ACORN vs. Ascon," Cryptology ePrint Archive 2019/184, 2019.

[21] NIST, *Lightweight Cryptography: Project Overview*, https://csrc.nist.gov/projects/lightweight-cryptography, 2019.

[22] ARM, *AMBA: The Standard for On-Chip Communication*, https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications, 2019.

[23] E. Trichina, "Combinational Logic Design for AES SubByte Transformation on Masked Data," Cryptology ePrint Archive 2003/236, Nov. 2003.

[24] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold Implementations Against Side-Channel Attacks and Glitches," in *Information and Communications Security, ICICS 2006*, ser. LNCS, vol. 4307, Springer Berlin Heidelberg, 2006, pp. 529–545.

[25] F. Farahmand, A. Ferozpuri, W. Diehl, and K. Gaj, "Minerva: Automated hardware optimization tool," in *2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017*, Cancun: IEEE, Dec. 2017, pp. 1–8.

[26] Cryptographic Engineering Research Group (CERG) at George Mason University, *Authenticated Encryption FPGA Ranking*, https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/rankings_view, 2019.