FELICS – Fair Evaluation of Lightweight Cryptographic Systems

Daniel Dinu*, Alex Biryukov, Johann Großschädl,
Dmitry Khovratovich, Yann Le Corre, Léo Perrin*

University of Luxembourg
{dumitru-daniel.dinu, alex.biryukov, johann.groszschaedl,
dmitry.khovratovich, yann.lecorre, leo.perrin}@uni.lu

Abstract. In this paper we introduce FELICS, a free and open-source benchmarking framework designed for fair and consistent evaluation of software implementations of lightweight cryptographic primitives for embedded devices. The framework is very flexible thanks to its modular structure, which allows for an easy integration of new metrics, target devices and evaluation scenarios. It consists of two modules that can currently asses the performance of lightweight block and stream ciphers on three widely used microcontrollers: 8-bit AVR, 16-bit MSP and 32-bit ARM. The metrics extracted are execution time, RAM consumption and binary code size. FELICS has a simple user interface and is intended to be used by cipher designers to compare new primitives with the state of the art. The extracted metrics are very detailed and assist embedded software engineers in selecting the best cipher to match the requirements of a particular application. The tool aims to increase the transparency and trust in benchmarking results of lightweight primitives and facilitates a fair comparison between different primitives using the same evaluation conditions.

Keywords: fair evaluation, benchmarking framework, lightweight cryptography, accurate measurements, comprehensive results

1 Introduction

The imminent expansion of the Internet of Things is creating a new world of smart devices in which security implications are very important. If we consider that brain stimulator circuits and heart pacemakers may be directly connected to a network to provide physicians with useful information in establishing and adjusting the therapy without physical examination of the patient, security plays a crucial role since unauthorized access to these critical devices can be life-threatening. The health sector is just one of the domains where the number of IoT devices is expected to grow significantly. Other IoT applications include supply chain management, smart homes, green cities and many more.

Besides the security aspects, the IoT introduces new challenges in terms of energy and power consumption. Thus the lightweight cryptographic primitives designed for IoT enabled devices must consume fewer resources, while providing the claimed level of security. In the recent past, the research community's interest for lightweight cryptography increased and as a result many lightweight algorithms were designed and analyzed from the security perspective. The academic activity in the last 10 years is quantified in around 1400 papers on lightweight cryptography according to [30]. The implementation effort focused on selecting the best design constructions in order to reduce the resource consumption, evaluating the performance figures achieved by hardware and software implementations on different platforms, and analyzing and improving the protection against side channel attacks.

Looking back at NIST contests for the selection of new cryptographic standards [27,26], we can see that weak designs from security perspective were disqualified after the first evaluation phase. In the following stages, the remaining algorithms had similar security margins and thus new evaluation criteria were necessary. This is the moment where hardware and software evaluation of the candidates plays a very important role. As is pointed in [16], the final ranking of candidates is closely related to the hardware and software performance figures. Since benchmarking frameworks allow for consistent evaluation, they are important not only in the selection process of new cryptographic standards, but also for fair comparison of ciphers' performances in given usage scenarios.

This year, NIST organizes a workshop [25] on lightweight cryptography to discuss the security and resource requirements of applications in constrained environments and potential future standardization

* The authors are supported by the CORE ACRYPT project funded by the *Fonds National de la Recherche* (Luxembourg)

of lightweight primitives. Considering the increasing market of IoT devices and the industry's need for a standard to secure IoT applications, tools designed to extract the performance figures of lightweight primitives on different platforms under the same conditions are required. These tools help cryptographers to evaluate proposed designs with respect to previous ones and can be used to break the tie between the candidates in the subsequent phases of the selection process.

Our Contribution Firstly, we analyze previous benchmarking frameworks to identify the strengths and weaknesses of each one. We formulate a set of design goals to ensure a fair evaluation of lightweight primitives on different platforms in the same conditions. Then, we describe the framework structure, extracted metrics and target devices. For each of the extracted metrics and for each supported device we describe the methodology and tools used. Based on the framework design and development experience, we provide a list of common errors and indications on how to avoid and manage them.

FELICS (***F**air **E**valuation of **Li**ghtweight **C**ryptographic **S**ystems*) [11] is a free, open source and flexible framework that assesses the performance of C and assembly software implementations of lightweight primitives on embedded devices. Thanks to the modular design, the framework can easily accommodate new metrics, usage scenarios or target devices. It is the core of an effort to increase transparency in lightweight algorithms' performances and aims to facilitate fair comparison of the assessed algorithms. Thus we are committed to maintaining a web page [11] where the tool can be downloaded and the assessed primitives results can be found.

To the best of our knowledge, this is the only free and open source benchmarking framework designed for fair and consistent evaluation of software implementations of lightweight primitives on various IoT embedded devices in the same usage scenarios. As the IoT field is expected to have a major growth in following years, FELICS will help the research community and industry with fair and detailed performance figures of lightweight primitives.

Organization of the Paper The rest of the paper is organized as follows. In Section 2, we discuss previous work with focus on design principles of benchmarking frameworks and the evaluation methodology used. Then in Section 3 we present the motivation and goals of the proposed framework. The framework for ***F**air **E**valuation of **Li**ghtweight **C**ryptographic **S**ystems* (FELICS) is described in Section 4, while Section 5 provides a description of the target devices. Section 6 describes how each metric is extracted for each of the target devices and Section 7 gives some experimental results regarding the time required to extract the supported metrics. Finally, Section 8 concludes the paper and gives future work directions.

2 Related Work

Over the time, several benchmarking frameworks have been designed to ease the evaluation of cryptographic primitives on different hardware or software platforms. In addition to these benchmarking frameworks, survey and benchmarking papers [23,15,22,21,24] were published. In this section we describe the previous work that helped us in designing the proposed framework. For each project analyzed we present the design requirements and constraints, the extracted metrics and the methodology used to ensure a fair and consistent evaluation.

2.1 BLOC Project

The BLOC project [9] aims to study the design of block ciphers dedicated to constrained environments. During the project a paper [10] about the performance evaluation of lightweight block ciphers for wireless sensor nodes was published. The C implementations of the studied ciphers along with the source code used to extract the analyzed metrics are available for free. The target device is the 16-bit MSP430F1611 [20] microcontroller commonly used in sensor nodes.

The three metrics considered (execution time, RAM requirement and code size) are extracted for a set of 17 ciphers. The cycle count is measured using the cycle accurate simulator `MSPDebug`. The RAM requirement is given by the stack usage for running the encryption key schedule, encryption and decryption operations. The stack consumption is computed by debugging with `msp430-gdb` the program execution on `MSPDebug` simulator. Breakpoints are inserted at the beginning and at the end of the program execution and afterward the number of modified words in the memory is computed. The data required to store the cipher state, the master key and round keys are not included in the RAM requirement. The code size is given by the `text` section of the binary file and is extracted using the `msp430-size` tool. The

metric extraction is done automatically through Bash scripts and the results are exported into LaTeX tables similar to those used in the paper [10].

Analyzing the project source code, we inferred that the library has some major drawbacks. Firstly, the RAM requirement given in the paper and on the project web site is wrongly computed because the framework implementers assume that the `unsigned int` data type requires one byte instead of two on 16-bit MSP430F1611 [20] microcontroller. Thus the RAM requirement provided in the paper is half of the actual value. Secondly, the library is not flexible at all and it does not allow easy addition of new devices or metrics. The provided library does not have a set of requirements that each implementation should follow and there is no common interface for assessing the performance of the implemented ciphers. Without a clear evaluation methodology, reference implementations that process one block at a time are compared with bit sliced implementations that process several blocks in parallel. Thirdly, some implementations of the studied ciphers do not verify the test vectors (e.g. LBlock). We wrote a patch that fixes the identified issues and sent it to the authors of the project, but it was not applied yet to the public repository mainly because the project is stalled for more than ten months.

The project has the merit of being one of the first attempts to evaluate a set of lightweight block ciphers on an embedded device. It also contains one of the largest collection of lightweight ciphers implementations available for free.

2.2 eBACS Project

The eBACS (ECRYPT Benchmarking of Cryptographic Systems) [7] was designed during the ECRYPT II project to measure the speed of a wide variety of cryptographic primitives on personal computers and servers. It integrates features for measuring the execution speed of public-key systems (eBATS), stream ciphers (eSTREAM) and hash functions (eBASH). The developed framework, SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives), provides a large collection of cryptographic primitives implementations. The open and free source code of the framework is written in C with inline assembly, Bash and Python.

The project web page provides information on how to submit new implementations as well as how to collect data for existing implementations using the framework. The requirements that the implementation of a cryptographic primitive has to fulfill in order to be evaluated using the framework are very well described and ensure a consistent evaluation of all implementations across all considered target platforms.

The framework allows benchmarking of C, C++ and assembly implementations. It automatically compiles the source code using different compilers and compiler options. The cycle count metric is computed using inline assembly instructions for each of the supported platforms. Because execution time is the only metric extracted, the submitted ciphers implementations are optimized only for speed. The results are extracted for different input data lengths across all compilers and compiler options and saved in a database. In fact the database is just a text file that contains a line for each implementation evaluated. The line is formed by space separated words that allow to identify the measurement conditions and the result.

Although the framework has not been updated lately, it represents the first step to consistent evaluation of cryptographic primitives. Thanks to the fair and clear evaluation methodology, it has been used as source of inspiration for other similar projects. One of the framework's strengths is given by the large number of computers with different architectures and characteristics used for the result collection process, while the main shortcoming is that is able to extract only the execution time.

2.3 XBX Project

The XBX (eXternal Benchmarking eXtension) project [32] is an extension of SUPERCOP that allows benchmarking of hash functions on different microncontrollers. Besides extending the eBASH capabilities to microcontrollers, XBX extracts two more metrics for the analyzed hash implementations: binary code size and RAM consumption. The code size is obtained through static analysis of the generated binary file. The RAM requirement is the sum of stack consumption and static RAM requirement obtained from the application binary. The cycle count values are subject to measurement errors because they are not extracted directly from the target devices, but from the XBH [34]. Most of the benchmarked algorithms are taken from the SUPERCOP.

The SUPERCOP framework extension is written in C, Perl and Bash and uses the same interfaces for implemented algorithms and generated results as SUPERCOP. The hardware layer consists of XBD (eXternal Benchmarking Device) and XBH (eXternal Benchmarking Harness) that communicate with

each other using either I^2C or UART and digital I/O lines. The XBH is connected to the PC running the XBS (eXternal Benchmarking Software) using the Ethernet port. Where more compilers are available, XBX retains the SUPERCOP capability to benchmark the same implementation using different compilers and compiler options.

The XBX is the first project to unitary measure performances of software implementations of cryptographic primitives built for different embedded devices using the same evaluation methodology. The results given in the report [34] are gathered for eight different devices with 8-bit, 16-bit and 32-bit CPUs from all major vendors and they were used in the second round of the SHA-3 competition [26]. The project web page [33] is not maintained any more and some links are broken.

2.4 ATHENa Project

The ATHENa (Automated Tool for Hardware EvaluatioN) project [31] aims at fair, comprehensive and automated evaluation of cryptographic cores developed using hardware description languages such as VHDL and Verilog. The goal of the framework is to spread knowledge and awareness about good performance evaluation practices in order to make comparison of competing algorithms fairer and more comprehensive.

The open source benchmarking environment is described in [17]. It is inspired from the eBACS project [7] and consists of a set of scripts written in Perl and Bash aimed at automated generation of optimized results for multiple hardware platforms. The metrics considered are area, throughput, and execution time and the primary optimization target is throughput to area ratio.

The framework can be used under Windows or Linux operating systems and supports different target FPGA families from Xilinx, Altera and Actel. It allows running all steps of synthesis, implementation, and timing analysis in batch mode and performs automated optimization of results aimed at one of the three optimization criteria: speed, area, and throughput to area ratio. The generated results can be exported in CSV, Excel and LaTeX formats.

During the SHA-3 contest [26] the tool played an important role due to the comprehensive results generated and published [18]. Besides being used during the SHA-3 competition [26], the framework is ready for the evaluation of authenticated encryption candidates from the CAESAR competition [8] and preliminary results are available on the project web site [31]. We note that although it provides comprehensive performance figures, it does not require revealing the source code. While this decision is meant to protect intellectual property, it narrows the transparency of the results.

2.5 ECRYPT II AVR ATtiny45 Performance Evaluations

During the ECRYPT II project two papers [13,5] regarding the performance evaluation of block ciphers and hash functions with applications in ubiquitous computing on Atmel AVR ATtiny45 8-bit microcontroller were published. The implementations done in assembly language are available for free [14,4]. Although the authors of the two benchmarks formulate a list of common constraints to be able to compare the performances, some of the guidelines were not always followed.

The papers consider the following metrics: code size, RAM use and execution time. A combined metric is computed as the product of code size and execution time divided by the block size for block ciphers. For hash functions the combined metrics are given by the product of code size and execution time and the product of RAM use and execution time. For block ciphers the average energy consumption is computed integrating the measured current consumption. The energy consumption for all studied ciphers is strongly correlated with the cycle count values.

The tools and methodology used to extract the main metrics are not described. Although the common interfaces used for the evaluation of the implementations are provided, no scripts to help with the metric collection process are provided. The use of assembly language for the implementations of algorithms has the advantage of illustrating the lightweight aspects of the studied ciphers better than C implementations, but in the same time it limits the code portability.

3 Motivation and Goals

The published lightweight designs give different performance figures on different platforms and different evaluation conditions. The exact methodology used to extract the figures is usually unclear. Considering that the performance figures are usually reported for different devices and that the measured operations

and measurement conditions differ from paper to paper, it is very hard to use the given values to compare different designs.

The lack of comparative performance figures creates the need for a fair and consistent way of extracting performance figures for lightweight ciphers. The results obtained using the same assessment methodology can be used to compare different algorithms. Using the performance values, cipher designers can infer which design constructions are better on different architectures. In the same time, the results can help engineers to select the best cipher for a given use case.

If the first proposed lightweight ciphers were mainly geared for hardware efficiency, in the last years, we notice that the focus is moving now to lightweight ciphers designed for efficiency in software. This new design direction for lightweight ciphers reinforces the need of reliable and accurate performance figures.

FELICS was created to allow the comparison of C software implementations of lightweight ciphers on different embedded devices commonly used in the IoT. The key characteristics or design goals are:

Fair evaluation To ensure a fair evaluation a clear assessment methodology was formulated. The methodology clearly indicates which are the requirements that each implementation should follow and how each metric is extracted for each supported device. Although sometimes the methodology can be considered restrictive, it is created to ensure a fair evaluation for every implementation.

Consistent evaluation The same methodology is used to assess the performance of all implementations of a given primitive type. Thus the evaluation is consistent across all the target embedded devices for all studied usage scenarios. The consistent evaluation allows easy comparison of the performance figures between the similar implementations of ciphers. It also facilitates the correct ranking of the ciphers' implementations using different criteria.

Accurate measurements The framework must provide accurate measurements. To achieve this goal, the tools used to extract the metrics and the measurement conditions must be precise. The simulators must be cycle accurate and the measurements on the boards must be carefully calibrated.

Free To ensure a widespread utilisation, the source code of the benchmarking framework and the source code of the evaluated implementations are available for free [11]. The source code allows to analyze and understand the link between the performance figures and implementation aspects.

Open source To increase the trust in the measurements, the framework source code is open. Anyone can analyze the source code, can detect and fix coding bugs, or even collaborate to the tool development with new modules and features.

Transparent The framework aims to ensure full transparency on how the metrics are collected for each implementation. Thus, besides publishing the source code of the evaluated implementations, the results obtained for each implementation are published on our web page [11].

Comprehensive results The extracted metrics are very detailed and aim to help understanding how different parts of an algorithm implementation are affecting the performances. The embedded software engineers can use the comprehensive results to select the best trade-offs for a specific use case.

Flexible The framework uses a modular architecture that facilitates further development. FELICS is designed to allow future development of new modules for assessing other types of cryptographic primitives. It also allows integration of new target devices and metrics. The process of integrating a new cipher implementation is very easy and it can be done following the methodology and requirements of the framework.

Automatic evaluation The framework is able to verify if an implementation follows the formulated requirements. It can automatically check if the implementation verifies the test vectors provided by the implementer for all target devices. The process of collecting the performance figures is completely automated and can be done in batch mode. The user can extract the results for a given list of ciphers and for a given list of architectures.

4 Benchmarking Framework

4.1 Structure

FELICS is written in C with inline assembly, Bash and Python and is designed to work on Linux operating systems. It allows benchmarking of C and assembly implementations that follow a given set or requirements. The C programming language was selected because of its widespread adoption and portability. If we consider that usually the reference implementations are provided in C language, then it is the natural choice to increase the number of targeted users. In the same time, the framework can target multiple embedded devices used in the IoT context using a single implementation, thus a limited

effort. FELICS also facilitates benchmarking of highly optimised assembly implementations which are platform dependent.

The usage scenarios are written in C, while the ciphers implementations can be written in C or assembly. Each module has a makefile that can build an implementation for a given architecture and scenario. The framework contains a collection of Bash (Bourne Again SHell) scripts that allow to fully automate the metric extraction process. Python scripts were used to perform operations that were too complicated or impossible to be done in Bash. FELICS is able to automatically generate the binary code, to check the implementation correctness using the provided test vectors and to extract the implementation metrics for the supported devices.

The current version of the framework includes the *Core Module*, a module for evaluating lightweight block ciphers and a module for assessing the performances of lightweight stream ciphers. Thanks to the modular structure depicted in Figure 1, FELICS can be easily extended with new modules capable to measure other primitives. Each module uses the features of the *Core Module* and in the same time provides it with scripts for batch processing.

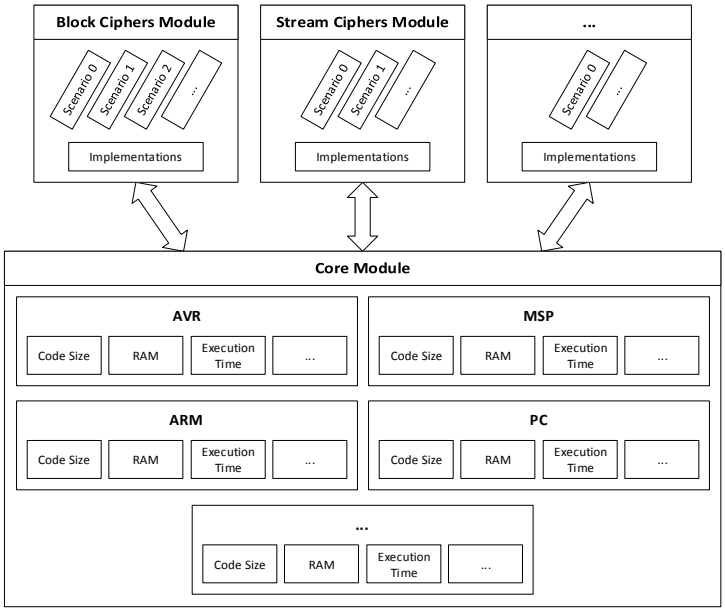

Fig. 1. Modular structure of FELICS

4.2 Core Module

The *Core Module* is the heart of the framework and provides the tools necessary to collect the metrics for each of the supported devices. The complete list of used tools and tool versions organized by extracted metric for each supported device is given in Table 1. The role of the module is to facilitate integration of new target devices and extracted metrics. It allows the user to collect the results for one or more of the modules that are integrated in the framework.

In addition to the supported embedded devices, the module gives the possibility to debug and evaluate cipher implementations built for personal computers. This feature is mainly added to reduce the complexity of the implementation and integration process and to ease the task of users.

In order to achieve the described design goals, each module formulates a specific set of requirements that every implementation should follow. Even though the requirements create additional constraints and limit the possibility to benchmark highly optimised implementations, they ensure a fair and consistent evaluation across all implementations.

Each module has a configuration file, `conf.sh`, that provides the module with information about the tools in the *Core Module* that can be used to extract the analyzed metrics for each target device. In the same time, each module should provide the `get_results.sh` script that can be called from the *Core Module* to extract the performance figures in batch mode.

Table 1. Tools used to extract the metrics for each target device

AVR	MSP	ARM
	Code size	
avr-size 2.23.1	msp430-size 2.21.1	arm-none-eabi-size 2.24.51
	RAM	
simavr a024f5	MSPDebug 0.22	J-Link GDB Server V4.94i
avr-gdb 7.6.50	msp430-gdb 7.2	arm-none-eabi-gdb 7.7.50
	Execution time	
Avrora 1.7.117	MSPDebug 0.22	Arduino Due board

4.3 Block Ciphers Module

The module allows the evaluation of lightweight block ciphers implementations. Each block cipher implementation has to use the function signatures from Listing 1.1 for the basic operations. In order to enable the framework to extract the metrics for each of the four operations, each operation has to be implemented in a separate C file. If the decryption key schedule is the same as the encryption key schedule, the decryption key schedule function has to be defined as an empty function. In the case the cipher does not have a key schedule, the encryption key schedule must be defined as a function that copies the master key into the round keys. The encryption and decryption operations are done in place to reduce the RAM consumption and the key should not be modified after running the key schedule.

Listing 1.1. Required function signatures for block ciphers

```
void RunEncryptionKeySchedule(uint8_t *key, uint8_t *roundKeys);
void Encrypt(uint8_t *block, uint8_t *roundKeys);
void RunDecryptionKeySchedule(uint8_t *key, uint8_t *roundKeys);
void Decrypt(uint8_t *block, uint8_t *roundKeys);
```

The block size, key size, round keys size and the number of rounds of the cipher have to be defined in the `constants.h` file. The constants used by the implementation should be declared in the same header file, while the definitions can be added to `constants.c` or any other `*.c` file, except for the predefined C files. The constants can be stored in Flash or RAM memory of the device and should be read with the corresponding macros. FELICS automatically checks if each implementation verifies the test vectors given in the `test_vectors.c` file.

The implementation information file, `implementation.info`, provides implementation details to the framework such that the common code and data to be considered just once when extracting the metrics. The implementers can split an implementation in as many files as they want if each implementation file is correctly listed in the `implementation.info` file. The implementation information file indicates if a key schedule is used for encryption and decryption.

A template cipher implementation and a file describing the module requirements is provided with the module to help users to integrate new implementations. The process of integrating an existing C or assembly implementation is thus very easy and consists in filling the functions of the template cipher with the cipher implementation following the rules described in the `README` file.

The module contains three evaluation scenarios, but can be easily extended with new evaluation scenarios. The cipher operation (Scenario 0) evaluates the basic operations performed by a block cipher. In this scenario a block of data is encrypted and then decrypted using the provided test vectors. The communication protocol (Scenario 1) assumes the encryption and decryption of 128 bytes of data using the CBC mode of operation. This scenario is suitable for secure communication in the IoT context and considers the limitations of IEEE 802.15.4 [19] and ZigBee [35] protocols used in sensor networks. The challenge-handshake authentication protocol (Scenario 2) is created to fulfill the authentication need in

the IoT context by using the block cipher in CTR mode to encrypt 128 bits of data. No key schedule is required because the cipher round keys are precomputed and stored in the device Flash memory.

Because the communication protocol and challenge handshake scenarios assume the encryption of 128 bytes and 128 bits of unpadded data respectively, the block size of the cipher in bits has to be equal to or divide 128.

4.4 Stream Ciphers Module

The performance figures of stream ciphers can be extracted for each stream cipher implementation that defines the functions described in Listing 1.2. The definition of each function has to be placed in a separate C file. The implementation of the encryption function must be able to process at least one byte. The encryption process is done in place to reduce RAM consumption. The cipher master key should not be modified after running the setup.

Listing 1.2. Required function signatures for stream ciphers

```
void Setup(uint8_t *state, uint8_t *key, uint8_t *iv);
void Encrypt(uint8_t *state, uint8_t *stream, uint16_t length);
```

The cipher state size, key size and initialization vector size have to be defined in `constants.h` file. The constants used by the stream cipher must be declared in `constants.h` file and defined in `constants.c` or any other `*.c` file, excepting the predefined `*.c` files. The implementer can choose to store the constants in Flash or RAM and has to use the corresponding macro to read the constants. The test vectors used by FELICS to check the correctness of the implementation should be defined in `test_vectors.c` file.

Integrating a new stream cipher implementation is very easy because the user is provided with a template implementation of a stream cipher and a file with implementation instructions. The implementer has just to fill the functions from the template cipher with the source code of the cipher according with the requirements described in the `README` file.

FELICS parses the `implementation.info` file to be able to count the common source code and constants only once in the extracted metrics. The implementation of each of the required functions can be split into several files provided that the implementation information is correctly given in the `implementation.info` file.

Two evaluations scenarios are implemented for this module, but new scenarios can be added at any time with minimal effort. The cipher operation (Scenario 0) is evaluated using the provided test vectors. The communication protocol (Scenario 1) is designed to secure the communication between wireless sensor nodes and consists in encryption of 128 bytes of data. Because the evaluation conditions are similar to the one used for block ciphers, these scenarios can also be used to compare the performance figures of block and stream ciphers.

4.5 Export Formats

The framework can export the extracted results for each scenario and target architecture in several formats in order to allow the user to analyze and post process the results. The supported formats are: raw data table, CSV file, XML file compatible with Microsoft Office Excel and LibreOffice Calc, MediaWiki table and LaTeX table. New formats can be easily added should the need arise. An archive with latest results in all mentioned formats is available for download on the FELICS web page [11]. On the same web page, a Python script for processing the CSV results can also be found. It allows ranking of existing ciphers implementations using the Figure of Merit (FOM) defined in [12], but can be easily modified to compute other values of interest.

4.6 Possible Sources of Errors

Next we briefly describe some of the issues that we encountered during the development and implementation of FELICS in order to help implementers to easily avoid or solve them.

- Most of the execution errors we encountered were due to the alignment issues. In order to fix these errors, the source of the problem has to be identified in the implementation code. Several ways to tackle the problem include rewriting the source code that generated the unaligned memory access, using the `ALIGNED` macro to align the data for each target devices, and using the `OPTIMIZATION_LEVEL_2`, `OPTIMIZATION_LEVEL_1` and `OPTIMIZATION_LEVEL_0` macros to reduce the optimization level of the function that fails.

– Implementers should have in mind the constraints of the target devices and must avoid using too much ROM and/or RAM to store huge look-up tables. If the RAM and/or ROM constants are not read using the correct macro, the results may be affected and the AVR implementation may crash.
– Special attention must be paid to implementations that crash on ARM evaluation board because they usually return nothing through the serial port connection.
– The values extracted on actual devices may vary depending on how the C code is translated into assembly instructions by the compiler, whether the accessed data are aligned or not, and which interrupts are enabled. For these reasons, the execution time of the same function may differ from scenario to scenario.

5 Target Devices

We selected three widely used target microcontrollers characterized by low power and energy consumption that are good representatives of most used 8-bit, 16-bit and 32-bit microprocessors in the Internet of Things context. Table 2 gives the main characteristics of the target devices used, while in the next paragraphs we provide a brief description of each one.

Table 2. Key characteristics of used microcontrollers

Characteristic	AVR	MSP	ARM
CPU	8-bit RISC	16-bit RISC	32-bit RISC
Frequency (MHz)	16	8	84
Registers	32	16	21
Architecture	Harvard	Von Neumann	Harvard
Flash (KB)	128	48	512
SRAM (KB)	4	10	96
EEPROM (KB)	4	-	-
Supply voltage (V)	4.5 - 5.5	1.8 - 3.6	1.6 - 3.6

5.1 8-bit AVR ATmega128 Microcontroller

The AVR ATmega128 [3] microcontroller manufactured by Atmel uses a CPU with RISC architecture and an on-chip 2-cycle multiplier. Most of the 133 instructions require a single cycle to execute. The rich instruction set is combined with the 32 8-bit general purpose registers (`R0 - R31`) with single clock access time. Six of the 32 8-bit registers can be used as three 16-bit indirect address register pointers (`X`, `Y` and `Z`) for addressing the data space. The instructions are executed within a two stages, single level pipeline: while one instruction is executed, the next instruction is pre-fetched from the program memory. Therefore, one instruction is executed every clock cycle. The Arithmetic Logic Unit (ALU) operations are divided into three main categories: arithmetic, logic and bit manipulation functions. All 32-bit registers are directly connected to ALU, allowing for two independent registers to be accessed in one instruction executed in one clock cycle.

It has a modified Harvard architecture where program and data are stored in separate physical memory regions located at different physical addresses. The separate memories and buses for program and data maximize the performance and parallelism. The memory includes 128 KB of Flash, 4 KB of SRAM and 4 KB of EEPROM. The data memory can be addressed using five different modes: direct, indirect, indirect with displacement, indirect with pre-decrement and indirect with post-decrement. An access to SRAM is performed in 2 CPU cycles.

Among the best microcontrollers when it comes to power consumption, Atmel ATmega128 provides a highly flexible and cost effective solution to many embedded control applications from building and home automation to medical and healthcare systems. It is working at supply voltages between 4.5 V and 5.5 V and has six different software selectable power modes of operation.

5.2 16-bit MSP430F1611 Microcontroller

The MSP430F1611 [20] microcontroler produced by Texas Instruments has a CPU with RISC architecture and 16 16-bit registers. Four of the registers are dedicated to program counter, stack pointer, status register and constant generator, while the remaining 12 registers (`R4 - R15`) are general-purpose registers. The 52 instructions with three formats (dual operand, single operand, jump) and seven addressing modes (register, indexed, symbolic, absolute, indirect, indirect auto-increment, immediate) can operate on byte and word data. The register to register operations take one clock cycle. The number of clock cycles required to perform an instruction depends on the instruction format and addressing mode used.

The Von Neumann memory of MSP430 has one shared address space for special function registers, peripherals, RAM and Flash memory. It includes 48 KB of Flash and 10 KB of SRAM. The Flash memory is bit, byte and word addressable and programmable.

Designed for low cost and low power consumption embedded applications, it requires a supply voltage between 1.8 V and 3.6 V and can reach a speed of 8 MHz. It has one active mode and five software selectable low-power modes of operation. Typical applications include industrial control, sensor systems and hand-held meters.

5.3 32-bit ARM Cortex-M3 Microcontroller

The Arduino Due board [1] uses the 32-bit Atmel SAM3X8 Cortex M3 [2] RISC CPU that executes Thumb-2 instructions. The instruction set allows high code density and reduced program memory requirements. The processor has a three level pipeline (instruction fetch, instruction decode and instruction execute) and 13 general purpose registers (`R0 - R12`).

The Harvard memory architecture includes 512 KB of Flash organized in two blocks of 256 KB and 96 KB of SRAM divided into two banks of 64 KB and 32 KB. The processor enables direct access to single bits of data in simple systems by implementing a technique called bit-banding. It supports two operating modes (thread and handler) and two levels of access for the code (privileged and unprivileged) enabling implementation of complex systems without sacrificing security.

Specifically designed to achieve high system performance in power sensitive embedded applications, such as microcontrollers, automotive body systems, industrial control systems and wireless networking, the processor operates at a maximum frequency of 84 MHz. The recommended supply voltage ranges between 1.6 V and 3.6 V.

6 Metrics

The tree metrics considered can be extracted in batch mode for a list of implementations, usage scenarios and target devices using the `collect_cipher_metrics.sh` script. We added support for these metrics because they outline the lightweight characteristics of the evaluated implementations. Derived or secondary metrics such as energy and power consumption were not included in the initial release, mainly because they are closely related to the basic metrics.

Detailed and accurate results are generated for each operation required by the corresponding module separately and for the all operations together. The comprehensive results can be used by embedded software engineers to decide what cipher operations should be implemented for a particular device and application. Where cycle accurate and free software simulators of the target embedded devices exist, they are preferred to the development boards because of usability reasons. While a software simulator can be downloaded and installed easily, a development board involves an acquisition and configuration cost. Next we describe how each metric is extracted on the considered target devices.

6.1 Code Size

The code size or binary code size measures the amount of information that is stored in the Flash memory of the target device. To extract the code size for each target device, the frameworks uses the GNU `size` tool, which lists the section sizes and the total size in bytes for a given binary file. The binary code size is given by the sum of the `text` and `data` sections. The `text` section of the binary file contains the code, while the `data` section stores global initialized variables, which are loaded from Flash into RAM at run time. The `bss` section of the binary file is not considered since the framework forbids the utilisation of global uninitialized variables. The code size of the `main` function where all operations are put together is not considered because it is the same for all studied ciphers.

The framework is able to determine the common parts using the implementation information file and uses them just once in the extracted code size value. Thus the computed program footprint is accurate and encourages code reuse.

FELICS uses `avr-size`, `msp430-size` and `arm-none-eabi-size` to extract the code size for AVR, MSP and ARM respectively. The exact versions of each tool is given in Table 1. The code size extraction process is completely automated and can be done using the `cipher_code_size.sh` script for a given cipher implementation and a given evaluation scenario.

6.2 RAM

The RAM consumption is split into stack requirement and data requirement. The stack consumption gives the maximum value of the RAM used to store local variables and return address after interrupts and subroutine calls. The data requirement represents the static RAM and is given by the size of the constants stored in target device RAM. It includes the data specific to each scenario such as data to encrypt, master key, round keys or initialization vectors.

The static RAM consumption is computed from the `data` section of the binary file using GNU `size`. As in the case of code size, using the implementation information file, FELICS considers the global initialized variables just once when they are used in several operations. The stack consumption is measured using the appropriate `gdb` client and the target device simulator or development board. At the beginning of each of the measured operations, immediately after the function call, the stack is filled with a memory pattern. At the end of the function execution, the memory content is compared with the initial pattern and the number of modified bytes gives the stack consumption. Since the measured functions do not use value arguments, there are no arguments saved on the stack that have to be taken into account. The measured operation return address is not considered since it is insignificant and the same for all ciphers on a given target device. The client and server tools used for computing the stack requirement are given in Table 1. The `cipher_ram.sh` script is able to extract the RAM requirement for a given cipher in a given usage scenario.

Another way to compute the stack requirement is to statically analyze the assembly instructions generated by the compiler and build the call graph for the measured function. For each entry in the call graph the maximum stack consumption is computed and stored. The stack usage of the measured function is given by the call path with the maximum stack requirements. This method is not able to solve recursive function calls and calls to functions from the standard C library. On the other hand using `gdb` client with a well tested simulator is less error prone than a tool developed from ground.

6.3 Execution Time

The execution time measures the number of CPU clock cycles spent on executing a given operation. The metric is extracted using either cycle accurate software simulators of the target microcontrollers or development boards.

The execution time is computed as the absolute difference between the system timer number of cycles at the end of the measured operation and at the beginning of the measured operation. To extract the number of cycles spent to execute the measured operations, FELICS simulates the cipher operation using the cycle accurate simulator `Avrora` [29,28] for AVR and the cycle accurate simulator `MSPDebug` [6] for MSP. For ARM, the framework inserts additional C and assembly code to read the system timer number of ticks at the beginning and at the end of each measured operation and then executes the program on Arduino Due [1] board. The measurement process on the ARM board was carefully adjusted to obtain accurate and precise results. We draw attention to the fact that extracted values for ARM may vary depending how the C code is translated into assembly instructions and how data is aligned in memory for different usage scenarios. Information about the used simulators are provided in Table 1. The `cipher_execution_time.sh` script extracts the execution time for a given cipher implementation and scenario.

7 Results

In Table 3 and Table 4 we give the average time to extract the studied metrics for each considered device and target scenario. The values are extracted using a computer with an Intel Core 2 CPU @ 1.4 GHz and 4 GB of memory running Ubuntu 14.04.2 LTS. The measurements include the time to build the cipher implementation for each target architecture and evaluation scenario, but do not include the time

to save the results in the supported formats. The average values are computed for one run of each metric extraction process over all implementations. The results are provided just to give an idea about the time required to extract the metrics and should not be seen as absolute or accurate values. In fact, the time required to extract each metric depends on many factors such as: the cipher implementation complexity, the tool used to extract the metric, the usage scenario, the characteristics and load of the computer that runs the framework, etc.

The extraction of the code size metric is the fastest on all devices because it consists only in analysing the binary files. The RAM metric collection takes more time than the extraction of execution time for AVR and MSP, mainly because of the interaction between the `gdb` client and the target device simulator. For ARM, the time required to extract the RAM requirement is very close to the time required to extract the execution time. The process of collecting the results from ARM device is more time consuming because it involves loading and running the program on the actual board. The communication between the target board and PC running FELICS also slows the metric extraction process.

From Table 3 we can infer that the process of collecting all the metrics for a block cipher for all target devices and usage scenarios requires around three minutes. Similarly from Table 4, the time required to collect all metrics for a stream cipher for all target devices and usage scenarios is less than two minutes.

The time required to extract the metrics in batch mode is 227 minutes for 86 implementations of block ciphers and 30 minutes for 24 implementations of stream ciphers. For each framework module, the given time values are for the three target devices (AVR, MSP and ARM) and for all usage scenarios. They also include the time required to export the results in all supported formats and are highly dependent on the computer load. In the case of batch processing, all implementations are built once at the beginning of the metric extraction process for a target architecture and usage scenario.

Table 3. Average time required to extract each metric for a block cipher implementation

	AVR			**MSP**			**ARM**		
	Code Size [s]	RAM [s]	Execution Time [s]	Code Size [s]	RAM [s]	Execution Time [s]	Code Size [s]	RAM [s]	Execution Time [s]
Scenario 0	0.85	3.78	1.54	1.05	10.85	1.06	1.38	15.53	16.40
Scenario 1	0.95	5.37	3.37	1.14	11.23	1.54	1.53	16.01	16.84
Scenario 2	0.97	3.61	1.68	1.13	8.22	1.11	1.54	13.54	15.82

Table 4. Average time required to extract each metric for a stream cipher implementation

	AVR			**MSP**			**ARM**		
	Code Size [s]	RAM [s]	Execution Time [s]	Code Size [s]	RAM [s]	Execution Time [s]	Code Size [s]	RAM [s]	Execution Time [s]
Scenario 0	0.39	3.05	1.23	0.38	7.57	0.40	0.51	11.27	13.18
Scenario 1	0.39	3.11	1.31	0.37	7.57	0.40	0.50	11.25	13.17

We do not provide performance figures of the implemented ciphers because the scope of the paper is to describe the framework and not the results obtained for a set of ciphers. For comprehensive results of around 100 implementations, we refer the reader to our web page [11].

8 Conclusion

In this paper, we introduced FELICS, a free and open source benchmarking framework for fair and consistent evaluation of cryptographic primitives. The motivation behind the framework design and

development is given by the lack of fair, comprehensive and consistent evaluation of lightweight cryptographic algorithms. Our aim is to increase the trust and transparency of results obtained by different algorithms and to ensure an independent environment for assessing performances of new designs. FELICS favors the comparison of performance figures between different ciphers due to the consistent evaluation methodology.

Currently the framework is able to benchmark lightweight block and stream ciphers on three different embedded devices. The metrics extracted for each device and evaluation scenario are: binary code size, RAM consumption and execution time. Thanks to its modular design, FELICS is very flexible and can be easily extended to benchmark new lightweight primitives, to extract new metrics, to collect the performance figures for other target devices or to evaluate the implemented algorithms in new usage scenarios. The framework source code together with implemented ciphers source code and performance figures are available on our web site [11]. We strongly encourage the community to contribute with implementations to the framework and support the trend for fair and transparent performance figures.

FELICS borrows and improves concepts from previous frameworks and, in the same time, adds new ideas and features. The result is a better framework for fair and consistent evaluation of cryptographic primitives. To the best of our knowledge, FELICS is the first benchmarking framework to evaluate lightweight primitives for the IoT context in different usage scenarios. It also provides full transparency in the performance figures by publishing the results and the corresponding source code on the project web site [11].

Possible new features include the addition of new modules to allow benchmarking of other cryptographic primitives (authenticated encryption, hash functions, public key cryptography), extraction of new metrics or support for new embedded devices. Implementing support for development boards where software simulators are used is another further development direction. Due to the free and open source properties of FELICS, anyone interested can contribute to the tool development with new features and bug fixes. User experience feedback and bug reporting will help to enhance the tool capabilities. Contributing with cipher implementations is another important direction, which will benefit the entire community.

Acknowledgments The authors thank Christian Wenzel-Benner and Jochen Gräf for providing the source code of the XBX [32] framework.

References

1. Arduino. Arduino Due. `http://arduino.cc/en/Main/arduinoBoardDue`.
2. Atmel. ARM Cortex-M3 datasheet. `http://www.atmel.com/Images/doc11057.pdf`.
3. Atmel. AVR ATmega128 datasheet. `http://www.atmel.com/images/doc2467.pdf`.
4. Josep Balasch, Barış Ege, Thomas Eisenbarth, Benoit Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, et al. Implementations of hash functions in Atmel AVR devices. `http://perso.uclouvain.be/fstandae/source_codes/hash_atmel/`.
5. Josep Balasch, Barış Ege, Thomas Eisenbarth, Benoit Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, et al. *Compact implementation and performance evaluation of hash functions in ATtiny devices*. Springer, 2013.
6. Daniel Beer. MSPDebug. `http://mspdebug.sourceforge.net/`.
7. Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to/`, February 2015.
8. CAESAR Competition. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. `http://competitions.cr.yp.to/caesar.html`.
9. Mickaël Cazorla, Sylvain Gourgeon, Kevin Marquet, and Marine Minier. Implementations of lightweight block ciphers on a WSN430 sensor. `http://bloc.project.citi-lab.fr/library.html`, February 2015.
10. Mickaël Cazorla, Kevin Marquet, and Marine Minier. Survey and benchmark of lightweight block ciphers for wireless sensor networks. In Pierangela Samarati, editor, *SECRYPT 2013 - Proceedings of the 10th International Conference on Security and Cryptography, Reykjavík, Iceland, 29-31 July, 2013*, pages 543–548. SciTePress, 2013.
11. CryptoLUX. FELICS - Fair Evaluation of Lightweight Cryptographic Systems. `https://www.cryptolux.org/index.php/FELICS`, 2015.
12. Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Lo Perrin, Johann Groschdl, and Alex Biryukov. Triathlon of lightweight block ciphers for the internet of things. Cryptology ePrint Archive, Report 2015/209, 2015. `http://eprint.iacr.org/`.

13. Thomas Eisenbarth, Zheng Gong, Tim Güneysu, Stefan Heyse, Sebastiaan Indesteege, Stéphanie Kerckhof, François Koeune, Tomislav Nad, Thomas Plos, Francesco Regazzoni, et al. Compact implementation and performance evaluation of block ciphers in ATtiny devices. In *Progress in Cryptology-AFRICACRYPT 2012*, pages 172–187. Springer, 2012.
14. Thomas Eisenbarth, Zheng Gong, Tim Güneysu, Stefan Heyse, Sebastiaan Indesteege, Stéphanie Kerckhof, François Koeune, Tomislav Nad, Thomas Plos, Francesco Regazzoni, et al. Implementations of low cost block ciphers in Atmel AVR devices. `http://perso.uclouvain.be/fstandae/lightweight_ciphers/`, February 2015.
15. Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6):522–533, 2007.
16. Kris Gaj. Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates using FPGAs. Presentation at CHES 2010, Santa Barbara, California, USA. Available for download at `http://www.chesworkshop.org/ches2010/presentations/CHES2010_Session06_Talk02.pdf`, August 2010.
17. Kris Gaj, J Kaps, Venkata Amirineni, Marcin Rogawski, Ekawat Homsirikamol, and Benjamin Y Brewster. Athena-automated tool for hardware evaluation: Toward fair and comprehensive benchmarking of cryptographic hardware using fpgas. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 414–421. IEEE, 2010.
18. Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Comparing hardware performance of fourteen round two sha-3 candidates using fpgas. Cryptology ePrint Archive, Report 2010/445, 2010. `http://eprint.iacr.org/`.
19. IEEE Standards Association. IEEE 802.15: WIRELESS PERSONAL AREA NETWORKS (PANs). `http://standards.ieee.org/about/get/802/802.15.html`.
20. Texas Instruments. MSP430F1611 datasheet. `http://www.ti.com/lit/ds/symlink/msp430f1611.pdf`.
21. Stéphanie Kerckhof, François Durvaux, Cédric Hocquet, David Bol, and François-Xavier Standaert. Towards green cryptography: a comparison of lightweight ciphers from the energy viewpoint. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 390–407. Springer, 2012.
22. Miroslav Knežević, Ventzislav Nikov, and Peter Rombouts. Low-latency encryption–is lightweight= light+ wait? In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 426–446. Springer, 2012.
23. Yee Wei Law, Jeroen Doumen, and Pieter Hartel. Survey and benchmark of block ciphers for wireless sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 2(1):65–93, 2006.
24. Mitsuru Matsui and Yumiko Murakami. Minimalism of software implementation. In *Fast Software Encryption*, pages 393–409. Springer, 2014.
25. National Institute of Standards and Technology (NIST). Lightweight Cryptography Workshop 2015. `http://www.nist.gov/itl/csd/ct/lwc_workshop2015.cfm`.
26. National Institute of Standards and Technology (NIST). SHA-3 Competition. `http://csrc.nist.gov/groups/ST/hash/sha-3/`.
27. NIST FIPS Pub. 197: Advanced encryption standard (AES). *Federal Information Processing Standards Publication*, 197:441–0311, 2001.
28. Ben L Titzer, Daniel K Lee, and Jens Palsberg. Avrora - The AVR Simulation and Analysis Framework. `http://compilers.cs.ucla.edu/avrora/`, 2005.
29. Ben L Titzer, Daniel K Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 67. IEEE Press, 2005.
30. Meltem Sönmez Turan. Challenges in Lightweight Crypto Standardization. Presentation at FSE 2015, Istanbul, Turkey. Available for download at `https://drive.google.com/file/d/0B0HadwwARB_-RTdOMzlGaDUwZDQ/view?pli=1`, March 2015.
31. George Mason University. ATHENa Project Website. `http://cryptography.gmu.edu/athena/`, 2015.
32. Christian Wenzel-Benner and Jens Gräf. XBX: eXternal Benchmarking eXtension for the SUPERCOP crypto benchmarking framework. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 294–305. Springer, 2010.
33. Christian Wenzel-Benner and Jens Gräf. XBX web page. `http://xbx.das-labor.org/trac`, February 2015.
34. Christian Wenzel-Benner, Jens Gräf, John Pham, and Jens-Peter Kaps. XBX benchmarking results January 2012. In *Third SHA-3 Candidate Conference (March 2012), http://xbx.das-labor.org/trac/wiki/r2012platforms_atmega1284p_16mhz*, 2012.
35. ZigBee Alliance. ZigBee Wireless Standard. `http://www.zigbee.org/`.