Triathlon of Lightweight Block Ciphers for the Internet of Things

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

SnT and CSC, University of Luxembourg
Maison du Nombre, 6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{dumitru-daniel.dinu,yann.lecorre,dmitry.khovratovich,leo.perrin}@uni.lu
{johann.groszschaedl,alex.biryukov}@uni.lu

Abstract. In this paper we introduce a framework for the benchmarking of lightweight block ciphers on a multitude of embedded platforms. Our framework is able to evaluate the execution time, RAM footprint, as well as binary code size, and allows one to define a custom "figure of merit" according to which all evaluated candidates can be ranked. We used the framework to benchmark implementations of 19 lightweight ciphers, namely AES, Chaskey, Fantomas, HIGHT, LBlock, LEA, LED, Piccolo, PRESENT, PRIDE, PRINCE, RC5, RECTANGLE, RoadRunneR, Robin, Simon, SPARX, Speck, and TWINE, on three microcontroller platforms: 8-bit AVR, 16-bit MSP430, and 32-bit ARM. Our results bring some new insights to the question of how well these lightweight ciphers are suited to secure the Internet of Things (IoT). The benchmarking framework provides cipher designers with an easy-to-use tool to compare new algorithms with the state-of-the-art and allows standardization organizations to conduct a fair and consistent evaluation of a large number of candidates.

Keywords: IoT, lightweight cryptography, block ciphers, evaluation framework, benchmarking.

1 Introduction

The Internet of Things (IoT) is a frequently-used term to describe the currently ongoing evolution of the Internet into a network of smart objects ("things") with the ability to communicate with each other and to access centralized resources via the IPv6 (resp. 6LoWPAN) protocol [5]. Today, the two most important and widely noticed exponents of the IoT are RFID technology (which has become a major enabler of modern supply-chain management and industrial logistics) and Wireless Sensor Networks (WSNs), which have found widespread adoption in several application domains ranging from home automation over environmental surveillance and traffic control to medical monitoring. A recent white paper by Cisco IBSG estimates up to 50 billion devices being connected to the Internet by the year 2020 [28], which implies that, in the not so distant future, every person in the developed world will be surrounded by dozens of sensors, actuators, RFID tags, and many other kinds of smart objects yet to be developed. This evolution from the Internet of people to the Internet of things will have a huge impact on our daily life and change the way we interact with the physical world surrounding us [5]. However, it is also evident that 50 billion smart devices connected to the Internet introduce unprecedented challenges to the security and privacy of their owners or users.

It is widely accepted that symmetric-key cryptosystems play a major role in the security arena of the IoT, but they need to be designed and implemented efficiently enough to not exhaust the scarce resources of typical IoT devices. Gligor defined in [30] lightweight cryptography as *cryptographic primitives, schemes and protocols tailored to extremely constrained environments such as sensor nodes or RFID tags*. A standard sensor node (e.g. the MICAz mote) is equipped with an 8-bit microcontroller (e.g. the ATmega128) clocked at 7.8 MHz and features 4 kB of RAM. Passive RFID tags do not even have a (software-programmable) processor, which means that performing cryptography on such tags is only possible through hardware implementation. The efficient implementation of cryptographic primitives so that they become applicable in the constrained regimes of sensor nodes and RFID tags is a challenging task since, for example, performance is conflicting with other metrics of interest like silicon area and power consumption (in the case of hardware implementation) or memory consumption and code size (for software). In addition, lightweight primitives should be able to withstand all known cryptanalytic attacks (e.g. linear and differential cryptanalysis when thinking of block ciphers) since lightweight cryptography is not meant to be "weak" cryptography in the sense that a lightweight primitive should not be the weakest link in the security of a system [30].

In this paper we present a survey of lightweight block ciphers along with software benchmarking results obtained on embedded 8, 16, and 32-bit microcontrollers. We consider three metrics of interest: execution

time, memory (i.e. RAM) requirement, and binary code size. To ensure a fair and consistent evaluation, we developed a benchmarking toolsuite that we make available to the cryptographic research community following the spirit of the well-known and widely-used eBACS system [9]. Our benchmarking tool is "open" in various aspects; first, it is possible to upload implementations of new ciphers as well as new (i.e. improved) implementations of ciphers that are already included. Second, the tool was developed from the ground up with the goal of supporting a wide range of embedded platforms through both cycle-accurate instruction set simulation and actual measurements on development boards. Currently, our tool includes cycle-accurate instruction set simulators for AVR ATmega and TI MSP430, as well as an ARM development board equipped with a Cortex-M3 processor[1]. We use GCC for all these platforms, but other compilers could be supported as well. Third, our tool is also open with respect to the evaluation metrics. Currently, it can evaluate three basic metrics, namely execution time, RAM footprint, and binary code size. Other metrics can be derived thereof or are, at least, closely related. For example, the energy consumption of a block cipher executed on an embedded processor operating in a certain power mode can be estimated by the product of execution time, supply voltage, and average power dissipation. However, since our framework supports development boards, it could be extended to acquire more accurate energy figures by simply measuring the processor's power dissipation while it executes a cryptographic algorithm.

Our benchmarking toolsuite accepts source codes written either in "pure" ANSI C or in C with inlined Assembly sections for the three processor architectures mentioned above. In this way, the toolsuite supports various trade-offs between performance and portability. At one end of the spectrum are highly-optimized implementations for which the complete encryption/decryption function consists of hand-crafted Assembly code. Assembly programming allows one to fully exploit the architectural features of a processor and, in this way, reach peak performance. The speed-up due to the integration of hand-crafted Assembly code is especially pronounced if a cipher performs a large number of operations that are significantly less efficient in C than in Assembly language (e.g. multi-word arithmetic, certain bit manipulations). Benchmarking results obtained from carefully-optimized Assembler implementations played an important role in the evaluation of candidates for cryptographic standards like the AES [43] and SHA-3 [45], and this will also be the case for future standardization activities in the area of lightweight cryptography for the IoT [44]. However, an implementation of a cipher written in Assembly language is architecture-dependent and, consequently, not portable. At the other end of the performance-portability spectrum are pure C implementations, which are highly portable but, in general, less efficient than their hand-crafted Assembly counterparts.

While the importance of benchmarking hand-optimized Assembly implementations is out of dispute, we argue that it makes also sense to benchmark portable C implementations of lightweight ciphers. Our argument is twofold and based on the specific properties and constraints of the IoT. First, it has to be noticed that there is no single dominating hardware platform in the IoT, in contrast to the "conventional" Internet of commodity computers, where the Intel architecture has a market share of over 90%. In fact, the IoT is populated by billions of heterogenous devices with largely incompatible processors and different operating systems. Supporting a large number of platforms with optimized Assembly code is tedious and error-prone since, for each processor architecture, a separate code base needs to be written, tested, debugged, and then maintained. In the light of ever-increasing time-to-market pressure, cryptographic engineers may value the portability of C code more than the performance of Assembly code. Our second argument is related to the steadily increasing research interest in lightweight ciphers with new designs being published (almost) every month. Implementations written in C often serve as proof-of-concept in the design phase of a new primitive to explore e.g. different candidates for a round function. Benchmarks generated from C implementations allow cipher designers to quickly evaluate the impact of various design options (e.g. round function, number of rounds) on execution time, RAM footprint and code size. In this way, designers can already assess in an early phase of the design cycle how a new primitive may compare with the state-of-the-art.

We report detailed benchmarking results for a total of 19 lightweight block ciphers, namely the AES [43], Chaskey [42], Fantomas [31], HIGHT [35], LBlock [58], LEA [34], LED [32], Piccolo [50], PRESENT [12], PRIDE [1], PRINCE [13], RC5 [48], RECTANGLE [61], RoadRunneR [6], Robin [31], Simon [7], SPARX [23], Speck [7], and TWINE [52]. Our rationale for selecting exactly the mentioned 19 ciphers is twofold; first, each of these candidates has some special property or feature that makes it interesting for applications in the IoT. Second, they cover a wide range of different design strategies and approaches. Our evaluation considers two application scenarios or use cases; the first relates to the encryption of messages transmitted in a Wireless Sensor Network (WSN) and the second is a simple challenge-response authentication protocol with applications in e.g. object identification or access control. To accommodate the different requirements

[1] The main reason for evaluating the execution time for ARM on a development board is that we could not find a cycle-accurate Cortex-M instruction set simulator of good quality that is freely available.

of these application scenarios, we evaluated at least two versions of most of the 19 ciphers, including a low-memory variant and a speed-optimized variant. The former can be seen as a "minimalist" implementation that favors low memory footprint and small code size over performance. On the other hand, the second implementation includes certain optimizations that increase code size and/or memory footprint (e.g. partial loop unrolling, use of small lookup tables) with the goal of improving performance. Roughly half of the implementations were written from scratch by us, whereby we put a comparable effort into optimizing each cipher to ensure a consistent and fair evaluation. The other half was either taken from other open-source projects or contributed by the designers of the algorithms or by volunteers; in all these cases we carefully reviewed the source codes and further optimized them whenever possible. In this way, we tried to minimize the impact of varying programming skills and/or experience. Most of our implementations are faster or on par with the best execution times reported in the literature on the platforms we consider. Therefore, these implementations form a solid code base for the benchmarking of lightweight block ciphers.

Related Work. There exist a number of related research projects that evaluate software implementations of lightweight block ciphers, but none of them analyzed execution time, RAM footprint, and code size on different 8, 16, and 32-bit platforms. We studied previous benchmarking initiatives in detail, but in the end we decided to develop a new benchmarking framework from scratch instead of contributing to an existing one since each of the existing frameworks has a certain issue or limitation that would have been difficult to fix. Nevertheless, understanding the strengths and weaknesses of other benchmarking initiatives helped us to design a flexible and powerful framework capable of collecting accurate and detailed results about the execution time, RAM consumption and code size of lightweight ciphers.

In the course of the BLOC project [17], a total of 16 lightweight block ciphers were evaluated on an MSP430 microcontroller. The provided C library [16] shows that the project does not insist on a common interface for all ciphers and there seems to be no way to easily integrate new platforms. By inspecting the benchmarking code we discovered a bug in the calculation of the RAM requirements because the authors assumed that a variable of type `unsigned int` has a size of one byte instead of two. Furthermore, some implementations did not verify the test vectors provided in the cipher specification[2]. For the block ciphers considered both in our paper and in [17], our results on MSP430 are, on average, three times better.

During the ECRYPT II project, a survey paper [25] with the results of a performance evaluation of 12 low-cost block ciphers on an 8-bit AVR ATtiny45 device was published. Among the analyzed ciphers are only designs that were introduced before 2012, i.e. more recent ciphers, such as Simon and Speck [7], are not included. The authors describe their evaluation methodology and the implementation guidelines they followed to ensure a fair comparison of the 12 lightweight ciphers. Although the Assembly source codes are available [27], there is no framework provided that can help users to asses the performance of new designs under the same conditions. The Assembly implementation results of this survey on AVR ATtiny45 are on par with our Assembly implementation results on AVR ATmega128.

The XBX extension [57] to SUPERCOP [9] allows one to benchmark hash functions on embedded devices and adds two new metrics, namely RAM footprint and ROM consumption. Unfortunately, the framework is currently not maintained any more, but still worth mentioning because of the consistent evaluation across several platforms and the importance of the benchmarking results for the SHA-3 competition [45].

Our Contributions. First, we designed and implemented a framework for fair and consistent benchmarking of lightweight cryptographic primitives on 8, 16, and 32-bit processors. Our work is motivated by the lack of a well-accepted and widely-used tool that allows the cryptographic research community to analyze and compare the execution time, RAM requirements and code size of lightweight primitives on a range of embedded platforms. These three metrics can be extracted at a very detailed level for different operations (e.g. encryption, decryption, key expansion) through a well-defined API. We make the entire source code of our framework available under GPL version 3 to facilitate the establishment of a completely free and open benchmarking environment for lightweight cryptosystems. The source code can be downloaded from the CryptoLUX wiki [19].

Second, we survey 19 lightweight block ciphers and analyze, in particular, their suitability for software implementation on resource-restricted devices. This set of ciphers covers a wide range of different design principles and includes a number of recent proposals with highly interesting properties, e.g. Simon/Speck [7], Robin/Fantomas [31], and SPARX [23]. We collected between two and up to 24 implementations of each cipher to account for different trade-offs between execution time, RAM footprint, and code size. For nine

[2] The maintainers of the BLOC project merged our pull request on GitHub that fixed the mentioned issues, see `http://github.com/kmarquet/bloc/pull/2`.

out of the 19 ciphers we have not only C implementations, but also optimized Assembly code for the three platforms we consider. In total, our repository includes over 250 implementations, of which we developed roughly half from scratch and the rest we took over (and slightly modified) from other open-source projects or they were contributed by the cipher designers. The source code of all our implementations is available under GPL and can be downloaded from the CryptoLUX wiki[3].

Third, we present detailed timing, RAM consumption, and code size figures of all 19 ciphers, which we generated with the help of our benchmarking toolsuite. Furthermore, we define two typical usage scenarios that aim to resemble security-related operations commonly carried out by "real" IoT devices. The results we obtained shed some new light on the relative efficiency of lightweight block ciphers because (i) several of our implementations are much faster or smaller than that of other survey and benchmarking efforts, and (ii) we include a few designs that have been published only very recently. Since lightweight cryptography is a rapidly progressing area of research, we also maintain a web page [19] with the latest results, which gets automatically updated when users provide new implementations. Our framework allows the user to define a custom "Figure Of Merit" (FOM) according to which an overall ranking of a set of block ciphers can be assembled. The FOM metric can use different weight factors for execution time, RAM footprint, and code size, and may even consider (cryptanalytic) security aspects.

To the best of our knowledge, this paper is the first to analyze a broad range of lightweight block ciphers on different processors in a comprehensive and consistent fashion, taking into account the specific constraints and requirements of the IoT. Our results allow one to infer some interesting relations between cipher design principles and performance figures, and, in this way, contribute to a better understanding of how to design and implement lightweight block ciphers.

2 Benchmarking Framework

Most papers introducing a new block cipher report some kind of results of some kind of performance evaluation on some kind of platform using some kind of implementation. These results are then used by the authors to claim that the proposed cipher has some kind of "advantage" over existing ciphers or compares "favorably" with the state of the art. However, the practical relevance of such comparisons is questionable since it is not easily possible to take differences in the characteristics of the target platforms or differences in the simulation/measurement conditions into account. Consequently, it is difficult to assess the relative efficiency of the numerous proposals for lightweight ciphers in a fair and consistent fashion. This motivated us to develop a benchmarking toolsuite that allows for a unified evaluation of a large number of candidates by collecting accurate and comprehensive results for execution time, RAM footprint, and code size. The toolsuite is currently able to extract these metrics from implementations for 8-bit AVR, 16-bit MSP430, and 32-bit ARM Cortex-M processors, but other platforms could be supported as well. We make the full source code of the benchmarking framework available under GPL to facilitate its acceptance in the cryptographic research community and to maximize transparency in the evaluation of lightweight block ciphers.

We developed our framework with the goal of being easy to use, but we also aimed for high flexibility in order to support various optimization strategies for lightweight ciphers. Therefore, the benchmarking framework accepts implementations written in C, which can optionally contain inlined Assembly segments to speed up performance-critical operations. New ciphers typically come with a reference implementation in C that can be easily cross-compiled for the three platforms mentioned above. This allows cipher designers to quickly assess the performance of a new block cipher on different 8, 16, and 32-bit processors. Currently, the GNU Compiler Collection (GCC) is used for all three platforms, but our toolsuite could be easily extended with other compilers. The benchmarking framework applies different combinations of compiler switches to optimize the code generation and achieve best possible results. To ensure a fair and consistent evaluation of ciphers, each implementation has to adhere to a pre-defined Application Program Interface (API) and follow a set of guidelines to meet certain constraints. A detailed description of the framework requirements can be found in Appendix B.

As stated in the previous section, we consider benchmarking results obtained with C implementations to be useful for cipher designers and for cryptographic engineers who prefer portable C code over platform-optimized Assembly code. Since cipher designers tend to write reference implementations in ANSI C, the effort of evaluating a new cipher boils down to adapting the C source code to meet the requirements of the framework. However, benchmarks generated with C implementations do often not reflect the full potential of a lightweight cipher because ANSI C can not efficiently express multi-word arithmetic operations and

[3] All results reported in this paper are based on version 1.1.20 of the FELICS framework, which can be downloaded from `http://www.cryptolux.org/index.php/File:FELICS.zip`

certain bit manipulations. In addition, the quality of the C compiler (i.e. its ability to apply sophisticated optimizations) may impact the relative performance of lightweight ciphers. To mitigate these issues, and to serve cryptographic engineers who are primarily interested in high speed rather than high portability, the toolsuite supports the benchmarking of hand-optimized Assembly implementations for the three considered platforms. We had both C and Assembly implementations available for nine of the 19 lightweight ciphers we benchmarked; the remaining ten ciphers were evaluated using C source codes only. In total, we analyzed more than 250 different C and Assembly implementations of 19 lightweight block ciphers. We make the full source code of all implementations available under GPL to ensure the reproducibility of our results and, in this way, increase the transparency and trustability of our evaluation process.

2.1 Evaluation Metrics

The benchmarking framework is able to extract three primary metrics, namely execution time, run-time memory (i.e. RAM) consumption, and code size. We consider these metrics as "primary" because (i) they determine to a large extent how well a block cipher meets the constraints and requirements of the IoT, and (ii) they can not be derived from other metrics. Since reaching high performance is the main design goal of essentially any software-oriented cipher, it is clear that a benchmarking framework needs to be capable to evaluate the execution time in a comprehensive and precise fashion. In addition, other metrics, such as the energy consumption of a cipher, have a strong correlation with the execution time[4]. While most existing papers that present implementation results for a lightweight block cipher only report execution times, we consider it important to take also RAM footprint and code size into account. Many IoT devices are so constrained that they feature only a few hundred bytes of RAM and a few kB of flash memory, which means RAM footprint and code size are crucial criterions for cryptographic engineers when selecting a lightweight cipher. In addition, since it is not possible to optimize all three metrics simultaneously, one has to find a trade-off. For example, common approaches to reduce the execution time, such as loop unrolling or the use of a lookup table to speed up the round function, generally increase the code size or RAM consumption or both. Our benchmarking framework allows a cryptographic engineer to analyze such trade-offs and, in this way, determine the best optimization strategy for the requirements of the target application and the constraints of the target device. The execution time, RAM footprint, and code size of an implementation can be combined into a single number by defining a Figure-of-Merit (FOM), which makes it possible to rank different implementations. We describe in Subsection 4.1 a FOM that is basically a weighted sum of the three metrics across the three platforms we support.

Execution Time. The execution time is quantified through the number of clock cycles required for the execution of each of the four basic operations of a block cipher, namely encryption, decryption, encryption key schedule, and decryption key schedule (see Appendix B). As mentioned earlier, our framework supports instruction-set simulators as well as the acquisition of real measurements from development boards. Concretely, the cycle-accurate simulators Avrora [55,54] and MSPDebug [8] are used to evaluate the execution times for the 8-bit AVR and 16-bit MSP430 platform, respectively. On the other hand, the cycle counts on the ARM Cortex-M3 are determined with help of the system timer by reading the SysTick Current Value Register (`SYST_CVR`). This register gets decremented with each processor clock and allows for very precise measurement of elapsed cycle counts. The framework automatically inserts C code for reading the system timer immediately before and immediately after the operation to be measured and calculates the execution time as the difference of the two timer values. However, the obtained execution time may vary by a few cycles depending on how the compiler translates the C code for reading the timer into Assembly instructions in different contexts and how the data types are aligned in memory. Therefore, it is possible to get slightly varying execution times for one and the same operation in different usage scenarios. We collected the timings for ARM reported in Section 4 using an Arduino Due board [2] with a Cortex-M3 [3] processor.

RAM Footprint. The RAM footprint is determined by the size of the `data` section (which contains all static variables that are initialized to a non-zero value) and the maximum stack consumption. There is no need to take the `bss` section into account since the two usage scenarios we developed for the benchmarking of lightweight ciphers (described further below) operate without uninitialized static data. In addition, the heap is not used at all because the framework does not permit any dynamically-allocated variables. The

[4] One can get a rough estimate of the energy consumption by simply forming the product of execution time, average power consumption of the target processor, and supply voltage. More accurate energy figures could be obtained by extending the framework to support power measurements on microprocessor development boards.

amount of RAM occupied by the `data` section is determined with help of the `size` tool from the GNU Binutils collection. On the other hand, the maximum stack consumption of an operation or usage scenario is evaluated in the standard way with a so-called stack canary. At the beginning of the operation/scenario (i.e. directly after the function call for that operation/scenario), the free stack space is filled with a certain pattern. Then, at the end of the function's execution, the values in the stack area are compared with the pattern and the number of overwritten bytes gives the stack consumption. The parameters passed to the function under evaluation do not need to be counted since the arguments are placed in registers on all three platforms and not pushed on the stack. However, what is added to the RAM consumption is the size of the operation-specific or scenario-specific variables (e.g. byte-arrays for plaintext, key, round key, initialization vector, etc.), which are declared and defined in the `main` function.

Code Size. The code size is measured in bytes and quantifies the amount of storage an operation or usage scenario occupies in the non-volatile memory (e.g. flash memory) of the target device. It is evaluated by applying the `size` tool on the relevant object files generated by the compiler. This tool, which is part of the GNU Binutils, lists the size of the different sections of an object file or executable. In order to obtain the overall code size, the framework simply adds the size of the `text` and `data` sections of the relevant object files. The `text` section contains the executable machine instructions the compiler generated from the source code. On the other hand, the `data` section comprises all static variables that are initialized with a non-zero value. The content of this section is loaded from flash memory into RAM at the beginning of the program execution. As already mentioned above, the `bss` section is empty since none of the benchmarked operations and scenarios use any uninitialized static variables. It should also be noted that common code fragments (e.g. auxiliary functions used in both encryption and decryption) are counted only once when computing the overall code size. Hence, it makes sense for implementers to identify common functionality and put it into a single C function or procedure. However, we do not take into account the size of the `main` function (from where the functions for the basic operations like encryption or decryption are called) because it is the same for all ciphers and not relevant in the context of the studied scenarios.

2.2 Usage Scenarios

Besides the evaluation of the four basic operations of a block cipher (i.e. encryption, decryption, encryption key schedule, and decryption key schedule), the benchmarking framework also supports more advanced forms of assessment based on usage scenarios. A usage scenario should implement some common security service with practical relevance for the IoT and utilize the basic cipher operations. In this way, it is possible to obtain realistic benchmarking results that are meaningful in the real world. The results reported in Section 4 are based on two simple usage scenarios, which we describe below. Further usage scenarios can be easily added thanks to the modular design of the benchmarking framework.

Scenario 1: Communication Protocol. This scenario covers the need for secure communication between two IoT devices such as two sensor nodes in a WSN. It is assumed that the sensitive data is encrypted and decrypted using a lightweight block cipher in CBC mode of operation. Since standard communication protocols for the IoT, such as IEEE 802.15.4 [36] and ZigBee [62], are characterized by low transmission rates and small packet sizes, we assume the plaintext to have a length of 128 bytes (i.e. 1024 bits) in this scenario. There is no need for a padding scheme because the length of the plaintext is a multiple of both 64 and 128 bits, which are the two block sizes we consider in this paper. Furthermore, we assume that the master key resides in RAM and that the round keys (obtained through the operation for key schedule) are also kept in RAM for later use by the encryption or decryption operation. The plaintext and initialization vector for CBC mode shall also be in the device's RAM at the beginning of the execution. In order to reduce the RAM footprint, the encryption is performed in place, which means the plaintext gets overwritten by the ciphertext (and vice versa for decryption). However, the key schedule does not modify the master key.

Scenario 2: Challenge-Response Authentication. This scenario is inspired by a simple authentication protocol where an IoT device proves that it is in possession of a secret key by encrypting a challenge using a block cipher. In real-world settings, the IoT device can, for example, be an RFID tag (see e.g. [29]) or a smart card. In this scenario we assume that a lightweight block cipher is used in CTR mode to encrypt 128 bits of data. The device has the full round key stored in flash memory, which means there is no need to store the master key and also no key schedule operation has to be performed. Both the 128-bit plaintext to be encrypted and the counter value are held in RAM at the beginning of the execution. In order to reduce the RAM footprint, the encryption is done in place, i.e. the plaintext gets overwritten by the ciphertext.

2.3 Target Devices

The IoT is populated by billions of devices that are equipped with a highly diverse and largely incompatible range of hardware platforms. In fact, the microcontroller population of the IoT is much more heterogeneous than the processor population of commodity computers, where the Intel architecture enjoys a market share of over 90%. Since there is no single dominating platform in the IoT, it is essential that a lightweight block cipher achieves consistently good performance on a variety of 8, 16, and 32-bit microcontrollers. It is also essential that a benchmarking framework is capable to collect implementation results from a wide range of platforms. Our framework supports the AVR ATmega128 [4] as example for an 8-bit architecture, the TI MSP430F1611 [53] as representative for a 16-bit platform, as well as the ARM Cortex-M3 [3] as example for a 32-bit RISC machine. However, as stated in the previous section, the benchmarking framework can be easily extended to support further platforms. A brief description of the main characteristics of the three currently supported microcontrollers can be found in Appendix A.

3 Analyzed Ciphers

Since our aim is to contribute to a better understanding of the relation between basic design methodologies for lightweight ciphers and the resulting software performance on resource-limited IoT devices, we selected 19 ciphers that represent a wide variety of design approaches based on Substitution-Permutation Networks (SPNs) and Feistel Networks (FNs). A classical example for an SPN is the AES [43,21], but other designs for the S-box and the linear layer are possible, as demonstrated by PRESENT [12], Robin, and Fantomas [31]. The overall structure of an SPN-based cipher can also vary while still maintaining a round function consisting of an S-box layer and a linear layer: LED [32] adds key material every four rounds only, while PRINCE [13] implements a property called α-reflection, which minimizes the overhead for decryption on top of encryption. Furthermore, it is also possible to build an SPN using only modular Addition, Rotation, and XOR (ARX), as was done by the designers of SPARX [23]. An FN, on the other hand, can be designed by utilizing a small SPN as the Feistel function, as in LBlock [58] and Piccolo [50], or with simple arithmetic and logical operations, as in Simon [7] and ARX designs like HIGHT [35], RC5 [48], and SPECK [7]. These operations may be data-dependent like in RC5. A variant of the FN is the Generalized FN, which uses more than two branches. The way the branches are mixed at the end of each round can consist of a simple rotation (HIGHT) or a dedicated permutation optimizing diffusion (TWINE [52], Piccolo). A high number of branches allows the use of very simple Feistel functions like in TWINE and HIGHT.

Besides representing a wide variety of different design approaches, most of the 19 lightweight ciphers we selected for our study have a certain property or feature that makes them particularly interesting for use in the IoT. We intentionally did not restrict our selection to software-oriented ciphers and included some designs that were developed for efficiency in hardware, e.g. Piccolo, PRESENT, and PRINCE. As stated in the previous section, the device population of the IoT is very heterogenous and shows extreme differences in terms of computational capabilities and resources. Some devices are so constrained that cryptographic operations can only be implemented in hardware (e.g. RFID tags), while other devices are powerful enough to run cryptographic software at acceptable speed. Since all these devices should be able to interact and communicate securely with each other, they have to use one and the same cipher. In order to be suitable for the IoT, a lightweight block cipher needs to be efficient in both hardware and software. Thus, it makes sense to evaluate the software performance of hardware-oriented ciphers and vice versa. In the following, we give an overview of the 19 lightweight ciphers we selected for benchmarking and describe how they can be implemented in software. The main characteristics of the candidates are summarized in Table 1.

AES. The AES is standardized by the NIST and by far the most-widely used block cipher today. It has an SPN structure with an internal state of 128 bits represented in the form of a (4×4)-byte matrix. The `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey` functions operate on the cipher's state [43,21]. To date, the best single-key cryptanalysis of AES-128 is a meet-in-the-middle attack on seven rounds out of ten [22]. Size-optimized implementations of the AES put the S-box and the round constants in lookup tables since they occupy just slightly more than 256 bytes. The source code of our size-optimized implementation mostly follows the cipher pseudocode on all three considered architectures. Since T-tables are very large (4 kB for either encryption or decryption), we did not include such implementations.

Chaskey. The Chaskey cipher is based on the π permutation of the Chaskey MAC algorithm [42] that is currently considered for standardization by the ISO/IEC. Said π permutation is a generalized FN and uses ARX operations on 32-bit words. The cipher has an Even-Mansour structure, which means there is no key

Table 1. Overview of the 19 lightweight block ciphers considered in this paper. Block, key and round key sizes are expressed in bits. The security level is the ratio of the number of rounds broken in a single key setting to the total number of rounds.

Cipher	Year	Block size	Key size	Round key size	Rounds	Security level	Type	Target
AES	1998	128	128	1408	10	0.70	SPN	SW, HW
Chaskey	2014	128	128	0	8/16	0.87/0.43	Feistel	SW
Fantomas	2014	128	128	0	12	NA	SPN	SW
HIGHT	2006	64	128	1088	32	0.81	Feistel	HW
LBlock	2011	64	80	1024	32	0.72	Feistel	HW, SW
LEA	2013	128	128	3072	24	0.63	Feistel	SW, HW
LED	2011	64	80	0	48	NA	SPN	HW, SW
Piccolo	2011	64	80	864	25	0.56	Feistel	HW
PRESENT	2007	64	80	2048	31	0.84	SPN	HW
PRIDE	2014	64	128	0	20	NA	SPN	SW
PRINCE	2012	64	128	192	12	0.83	SPN	HW
RC5*	1994	64	128	1344	20	0.80	Feistel	SW, HW
RECTANGLE	2015	64	80/128	1664	25	0.72	SPN	HW, SW
RoadRunneR	2015	64	80/128	0	10/12	0.5/0.58	Feistel	SW
Robin/Robin*	2014	128	128	0	16	1/NA	SPN	SW
Simon	2013	64	96/128	1344/1408	42/44	0.71/0.70	Feistel	HW, SW
SPARX	2016	64/128	128	1600/4224	24/32	0.62/0.68	Feistel	SW
Speck	2013	64	96/128	832/864	26/27	0.73/0.74	Feistel	SW, HW
TWINE	2011	64	80	1152	36	0.64	Feistel	HW, SW

* We use RC5 with increased number of rounds, RC5-20.

schedule but the master key is simply XORed to the internal state before and after π is applied. Chaskey-LTS (Long Term Security) has twice as many rounds as Chaskey and is recommended as a fallback in the case of cryptanalytic breakthroughs. Currently, the best attack against Chaskey is a differential-linear attack on seven out of eight rounds [40]. We benchmarked the C implementation provided by the designers, which is straightforward thanks to the simple structure of the cipher. In addition, we developed implementations in Assembly language from scratch. The execution times of both can be improved by unrolling several rounds at the cost of larger code size.

Fantomas. Fantomas is a 128-bit cipher belonging to the family of LS-designs [31]. Its linear layer consists in the parallel application of so-called "L-boxes," while the S-box is designed to simplify the implementation of masking, a countermeasure against Differential Power Analysis (DPA). There is no key-schedule; the master key is simply added in every round. At the time of writing this paper, there was to our knowledge no attack against Fantomas. A software implementation of Fantomas usually combines lookup-table based L-boxes with bit-sliced S-boxes, which are computed using a Feistel structure. Storing the four 512 B L-boxes in RAM instead of flash improves the execution time by a quarter on AVR and ARM. Our implementations are based on the C source code provided by the designers.

HIGHT. The lightweight cipher HIGHT is a generalized FN with an ARX structure. More precisely, the Feistel functions perform only logical XOR and bitwise rotations. The output of the Feistel functions is combined with the other branches using either XOR or addition modulo 2^8 [35]. An impossible differential attack breaks 26 out of 32 rounds of HIGHT [46]. All implementations we benchmarked follow closely the specification from [33], which modifies the design of the original paper [35]. The 128 7-bit δ constants are either computed when the key-schedule function is called or pre-computed and stored in flash or RAM. An entirely unrolled version with inlined auxiliary round functions $\mathtt{F}_0$ and $\mathtt{F}_1$ requires only half of the cycles of the reference implementation. When implemented in Assembly language, the execution time decreases by 50% on MSP and by 10% on AVR and ARM, respectively.

LBlock. LBlock is an FN with 32 rounds. The Feistel function consists of a logical XOR with the round subkey, a substitution layer of eight different S-boxes, and a permutation of eight nibbles. Furthermore, the content of one of the branches is rotated by eight bits in each round. The chosen design trade-offs between security and performance led not only to hardware efficiency but also software efficiency [58]. To date, the

best cryptanalytic result is obtained through an impossible differential attack against 23 out of 32 rounds [14]. The benchmarked LBlock implementations follow the specification from [58]. Optimization strategies include performing operations on 8, 16 or 32 bits when possible, storing the S-boxes in flash or RAM, and unrolling the loops. The best execution time on ARM is achieved by the fully-unrolled implementation using 32-bit operations, with the S-boxes stored in RAM.

LEA. The block cipher LEA uses a generalized FN with four 32-bit branches [34]. Designed for high-speed software encryption on 32-bit platforms, the cipher can be efficiently implemented in hardware as well. The designers mention a boomerang attack against 15 rounds, which is, to our knowledge, the best cryptanalytic result to date. The benchmarked assembler implementations are based on three different optimization strategies: fast execution time, small code size, and a trade-off between speed and size. These optimizations are facilitated by LEA's simple structure requiring only 32-bit operations.

LED. The AES-based cipher LED is aimed at very compact hardware implementation while maintaining reasonable performance in software. It represents the state by a (4×4)-nibble matrix and uses similar round transformations as the AES, except that they are nibble-oriented. A distinguishing characteristic of LED is the absence of a key schedule; the round keys are simply replaced by a part of the master key [32]. To the best of our knowledge, there are no attacks on LED-80. However, there is a differential attack that covers 16/32 rounds of LED-64 and 24/48 rounds of LED-128 [41]. The structural attack breaking 32/48 rounds of LED-128 proposed in [24] is unlikely to be adaptable to LED-80. Our LED implementation combines the `SubCells`, `ShiftRows`, and `MixColumnsSerial` operations into a table lookup to reduce execution time.

Piccolo. Piccolo has a generalized FN structure with four 16-bit branches. To improve diffusion, Piccolo uses a byte permutation between rounds. Picoolo's Feistel function consists of two S-box layers separated by a diffusion matrix [50]. The currently best attack against Piccolo-80 is a meet-in-the-middle attack on 14 rounds, which was presented by the designers. Our Piccolo implementation follows closely the description provided in [50]. The arithmetic in $\mathrm{GF}(2^4)$ uses only XORs and two small lookup tables for multiplication by two and three. Both the S-box and the key schedule constants are stored in lookup tables. No specific loop unrolling is applied.

PRESENT. PRESENT has an SPN structure and comes with a bit-oriented permutation layer. The non-linear layer is based on a single 4-bit S-box that was designed for efficiency in hardware [12]. A truncated differential attack against 26 out of 31 rounds of PRESENT is described in [11]. Since the S-box is quite small, a lookup table is used in all our implementations. However, its combination with a bit permutation over a 64-bit word is difficult to optimize without introducing extremely large lookup tables (up to 1 MB for decryption). The size-optimized implementation resembles the cipher's pseudocode and was taken from [16]. In general, the bit-oriented design of PRESENT makes C implementations very slow unless one can afford huge lookup tables. Our Assembly implementations take advantage of bit-manipulation instructions that the target platforms support. On AVR, the Assembly implementation is around 12 times faster than its C counterpart, while the MSP Assembly version is even 19 times faster than the C code.

PRIDE. The block cipher PRIDE is an SPN with a strong linear layer and a bit-sliced S-box, which are optimized for 8-bit microcontrollers [1]. It uses the so-called FX construction with the same key for pre- and post-whitening and a different key as basis for the round keys. A differential attack on 19 out of 20 rounds is described in [59]. The designers contributed a C implementation using only 8-bit operations. PRIDE's simple key schedule can be performed on the fly to reduce the RAM requirements at the cost of execution time. The S-box requires only bitwise operations, and also the linear layer consisting of four transformations (one for every 16 bits of the state) can be implemented efficiently in software.

PRINCE. Similar to PRIDE, PRINCE is an FX construction, whereby the first two subkeys are used as whitening keys, while the third subkey is the 64-bit key for a 12-round SPN called PRINCE_{core}. PRINCE introduced the α-reflection property: encryption with a given key corresponds to decryption with a related key [13]. To date, the best cryptanalytic result is a multiple differential attack on ten out of the 12 rounds [15]. We implemented PRINCE as described in the original paper [13,15]. The optimization strategies we considered include the use of 8, 16, 32, and 64-bit operations where possible and different amounts of loop unrolling. We obtained the best performance with fully unrolled implementations based on 8-bit operations for AVR and 16-bit operations for MSP. On ARM, the best execution times were achieved using a partially unrolled version with 32-bit operations.

RC5. RC5 is an FN that uses data-dependent rotations [48]. Even though RC5 was designed long before lightweight ciphers became popular, it is obviously suitable for resource-constrained devices like wireless sensor nodes as shown in e.g. [47]. The block and key size, as well as the number of rounds, can be chosen freely. We use RC5-32/20/16, i.e. a version of RC5 operating on two 32-bit words with a total of 20 rounds (or 40 half-rounds) and a 16-byte key. The number of rounds was chosen so as to have a security margin of 0.80. RC5-32/12/16 can be attacked using differential cryptanalysis as explained in [10]. This attack can be extrapolated to 18 rounds, but would require almost the full codebook (i.e. 2^{64} ciphertexts). RC5 was implemented by slightly adapting the reference code provided in [48]. Because of its elegant and simple design, there are only very few possibilities for optimization. To explore different trade-offs, we unrolled the cipher's operations and pre-computed the encryption-key-schedule array `S` to store it in flash or RAM.

RECTANGLE. The block cipher RECTANGLE is an SPN that allows for efficient implementation in hardware and software thanks to its bit-sliced structure [61]. Its non-linear layer applies a 4-bit S-box to each column of the state, which is represented through a (4×16)-bit matrix, while the linear layer rotates each row by a different amount. A differential attack that covers 18 out of 25 rounds is described by its designers. RECTANGLE was implemented in C and Assembly by its designers using different optimization strategies. The bit-sliced S-box is relatively fast in software because it uses only logical operations. On the other hand, the simple linear layer involves three rotations of 16-bit words by 1, 12, and 13 bits, which can be efficiently implemented on 8, 16, and 32-bit architectures.

RoadRunneR. RoadRunnerR has an FN structure that was designed with the aim of high efficiency on 8-bit processors and provable security in terms of minimum number of active S-boxes in differential and linear trails [6]. The Feistel function is an SPN composed of four 4-bit S-box layers, three linear layers, as well as three key additions. There exists a high-probability truncated trail covering five rounds, which can be utilized to attack a 7-round variant of RoadRunneR-128 [60]. RoadRunneR facilitates implementations in a bit-sliced fashion and is easy to optimize thanks to its simple structure. The bit-sliced S-box and the linear layer use only bitwise operations and rotations of 8-bit values by 1 bit and are, therefore, very fast in software. To reduce the RAM requirements, the round keys can be computed on the fly.

Robin. Robin is a 128-bit cipher similar to Fantomas, but its "L-boxes" are involutions [31]. The lookup table-based diffusion layers and the structure of the S-boxes makes this family of ciphers good candidates for Boolean masking in bit-sliced software implementations. Unfortunately, there exists a set of weak keys of density 2^{-32} that leads to a practical attack on the full primitive as shown in [39]. In response to this so-called invariant subspace attack, the designers of Robin proposed Robin* [37], in which the 8-bit round constant is replaced by a 128-bit round constant. Robin was implemented in different ways based on the C code provided by its designers. The two L-boxes are stored in flash or RAM, while the S-box layer is computed at each round using the Feistel structure. Robin* requires more memory and is also slower than the original Robin due to the costly derivation of the 128-bit round constants.

Simon. Simon uses an FN structure with a very simple round function performing bitwise XOR, bitwise AND, and circular left-shifts. It was primarily optimized for high performance in hardware, but achieves excellent results in software as well [7]. Differential attacks on 30 out of 42 rounds of Simon-64/96 and on 31 out of 44 rounds of Simon-64/128 are presented in [18]. Optimized implementations of Simon written in Assembly (for AVR and MSP) and C (for ARM) were provided by its designers. Simon's simple structure enables various trade-offs between code size and execution time by combining a different number of rounds in one loop iteration.

SPARX. The block cipher SPARX is an SPN designed on the foundation of the recently introduced Long Trail Strategy (LTS), which allows the use of a large and relatively "weak" S-box rather than a small and strong one [23]. Its ARX-based S-box consists of an unkeyed Speck-32 round [7], while the linear layer is inspired by that of Noekeon [20]. The designers studied in [23] an integral attack based on Todo's division property covering 15 out of 24 rounds of SPARX-64/128 and 22 out of 32 rounds of SPARX-128/128. Due to its simple and flexible structure, SPARX can be implemented using various optimization strategies. We explored various different trade-offs between execution time and code size by unrolling the rounds of a step function and performing one or two step functions at once.

Speck. Speck was designed to achieve high efficiency in hardware and software, especially when executed on resource-restricted microcontrollers [7]. It uses a Feistel structure in which both branches are modified

at each round using bitwise XOR, modular addition, and circular shifts in both directions. The to-date best cryptanalytic results against Speck-64/96 and Speck-64/128 are differential attacks targeting 19 and 20 rounds out of 26 and 27, respectively [51]. Speck has a simple round function that is extremely fast and takes just a few bytes of code. Optimized implementations of Speck written in Assembly (AVR, MSP) and in C (ARM) were provided by the designers. Depending on the optimization strategy, one or several round functions can be unrolled to improve the execution time at the cost of a minor increase in code size.

TWINE. TWINE is a generalized FN with 16 branches of four bits [52]. The Feistel function just consists of a key addition and the application of a 4-bit S-box. TWINE's linear layer is a nibble permutation with a much higher diffusion than a nibble rotation as used in e.g. HIGHT. The designers aimed for both small footprint in hardware and low RAM as well as ROM/flash consumption in software. The best attack on TWINE-80 is a multi-dimensional zero-correlation linear attack on 23 out of 35 rounds [56]. Since TWINE has a rather minimalist structure, the speed-optimized implementation is only marginally larger than the size-optimized version. The speed-optimized implementation described by the designers in [52] places the 4-bit branches in separate bytes, which means the state becomes twice as large. We wrote a size-optimized implementation from scratch. Both implementations are small enough to run on all three platforms.

4 Benchmarking Results

In this section, we firstly describe our evaluation methodology, including the Figure Of Merit (FOM) we developed to rank the candidates, and then we present and discuss the benchmarking results of 19 ciphers in the two scenarios described in Section 2.2. A block size of 64 bits was used when available, otherwise we resorted to 128-bit blocks. We only consider cipher versions with a key length of at least 80 bits, which we deem the minimum security level acceptable for common IoT applications. For some ciphers we collected benchmarking results for 80 and 128-bit keys to assess how the key length impacts execution time, RAM footprint, and code size.

4.1 Evaluation Methodology

At the time of writing this paper, our repository contained between two and 24 different implementations of 19 lightweight ciphers, and more than 250 altogether. Using the tool suite introduced in Section 2, we benchmarked the implementations in a highly automated way on three platforms (AVR, MSP, ARM) and for two usage scenarios. It is possible to rank all these 250+ implementations according to their execution time, RAM footprint, or code size in any scenario on any platform. In addition, we maintain an interactive web-page [19] with up-to-date benchmarking results where all these ranking options can be chosen. Due to space limitations we can only present a small subset of the results in this paper, whereby we aggregated the data as described below. We also introduce a *Figure-of-Merit (FOM)* that allows us to assemble an overall ranking of the 19 evaluated ciphers.

As explained in Section 2, our benchmarking tool determines the execution time, RAM footprint, and code size of each implementation on each platform, which yields a massive amount of "raw" data. Then, for each implementation i and platform d, we calculate a *performance indicator* $p_{i,d}$ that aggregates the three metrics from $M = \{$ execution time, memory consumption, code size $\}$ according to the formula

$$p_{i,d} = \sum_{m \in M} w_m \frac{v_{i,d,m}}{\min_i(v_{i,d,m})}, \tag{1}$$

where $v_{i,d,m}$ is the value of metric m for implementation i on platform d; w_m is the *relative weight* of metric m and $\min_i(v_{i,d,m})$ represents the minimum value of metric m among all considered implementations of all considered ciphers on the same platform d. By default, we set $w_m = 1$ for each platform and select the implementation with the best (i.e. smallest) performance indicator $p_{i,d}$ on each platform for the calculation of the FOM. However, the benchmarking tool suite also allows one to choose different weights for the three metrics (which can be useful when e.g. execution time is more important than RAM footprint or code size) or the three platforms (when e.g. ARM is more important than AVR or MSP). Finally, for each cipher and the selected set of best implementations i_1, i_2, i_3 (one for each platform), we calculate the FOM value as the average performance indicator across the three platforms:

$$\mathrm{FOM}(i_1, i_2, i_3) = \frac{p_{i_1,AVR} + p_{i_2,MSP} + p_{i_3,ARM}}{3} \tag{2}$$

Table 2. Results for Scenario 1: Encryption and decryption of 128 bytes of data in CBC mode. For each cipher and each platform, the results of the implementation with the best performance indicator according to Equation (1) are shown. The Figure-of-Merit (FOM) is based on the performance indicators on all three platforms (the smaller the FOM value, the better the implementations of a cipher).

Cipher			**AVR**			**MSP**			**ARM**			
	Block [b]	Key [b]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	FOM
Chaskey	128	128	1328*	229*	**20622***	900*	222*	**16674***	**438**	**236**	**9851**	4.0
Chaskey-LTS	128	128	1328*	229*	33102*	904*	222*	25394*	**438**	**236**	12859	4.6
Speck	64	96	966*	294*	39875*	**556***	288*	31360*	492	308	15427	5.1
Speck	64	128	**874***	302*	44895*	572*	296*	32333*	444	308	16505	5.2
Simon	64	96	1084*	363*	63649*	738*	360*	47767*	600	376	23056	7.0
Simon	64	128	1122*	375*	66613*	760*	372*	49829*	560	392	23930	7.2
RECTANGLE	64	80	1152*	352*	66722*	812*	398*	44551*	664*	426*	35286*	8.0
RECTANGLE	64	128	1118*	353*	64813*	826*	404*	44885*	660*	432*	36121*	8.0
LEA	128	128	1684*	631*	61020*	1154*	630*	46374*	524*	664*	17417*	8.3
SPARX	64	128	1198*	392*	65539*	966*	392*	36766*	1200*	424*	40887*	8.8
SPARX	128	128	1736*	753*	83663*	1118*	760*	53936*	1122*	788*	67581*	13.2
HIGHT	64	128	1414*	333*	94557*	1238*	328*	120716*	1444*	380*	90385*	14.8
AES	128	128	3010*	408*	58246*	2684*	408*	86506*	3050*	452*	73868*	15.8
Fantomas	128	128	3520	227	141838	2918	222	85911	2916	268	94921	17.8
Robin	128	128	2474	229	184622	3170	238	76588	3668	304	91909	18.7
Robin*	128	128	5076	271	157205	3312	238	88804	3860	304	103973	20.7
RC5-20	64	128	3706	368	252368	1240	378	386026	624	376	36473	20.8
PRIDE	64	128	1402	369	146742	2566	**212**	242784	2240	452	130017	22.8
RoadRunneR	64	80	2504	330	144071	3088	338	235317	2788	418	119537	23.3
RoadRunneR	64	128	2316	**209**	125635	3218	218	222032	2504	448	140664	23.4
LBlock	64	80	2954	494	183324	1632	324	263778	2204	574	140647	25.2
PRESENT	64	80	2160*	448*	245232*	1818*	448*	202050*	2116*	470*	274463*	32.8
PRINCE	64	128	2412	367	288119	2028	236	386781	1700	448	233941	34.9
Piccolo	64	80	1992	314	407269	1354	310	324221	1596	406	294478	38.4
TWINE	64	80	4236	646	297265	3796	564	387562	2456	474	255450	40.0
LED	64	80	5156	574	2221555	7004	252	2065695	3696	654	594453	138.6

* Results for Assembly implementations.

The FOM defined by Equation (2) is an attempt to condense three performance indicators into a single overall performance figure. Using this FOM allows one to compare (and rank) different ciphers, taking into account two major requirements for the IoT, namely (i) that not only speed but also memory requirements and code size are important, and (ii) that good implementation results should be achieved on a wide range of platforms and not just a single one. Of course, there are many alternative ways to define a performance indicator or a FOM. The performance indicator specified by Equation (1) aggregates each of the three considered metrics in relation to the best (i.e. smallest) value of the metric among all implementations. One could, for example, also relate the value of the memory and code-size metric to the amount of resources available on a device and calculate the performance indicator $p_{i,d}$ as

$$p_{i,d} = w_{\text{time}} v_{i,d,\text{time}} + \sum w_m \frac{v_{i,d,m}}{\max_i(v_{i,d,m})} \quad \text{for } m \in \{\text{memory}, \text{size}\}. \tag{3}$$

where $\max_i(v_{i,d,m})$ is the total RAM capacity (for the memory metric) or the total flash capacity (for the code-size metric) of device d; see Appendix A for details. In this way, we essentially measure the fraction of the totally available resources occupied by the implementation of a cipher. The FOM could, besides the efficiency metrics that determine the performance indicator, also take security aspects into account; the Figure of Adversarial Merit (FOAM) proposed in [38] serves as a good example.

4.2 Discussion of Results

Table 2 shows the results for Scenario 1 ("Communication Protocol") ordered by FOM value, whereby we measured the encryption and decryption of 128 bytes using CBC mode, including key schedule. The top-3

Table 3. Results for Scenario 1: Encryption and decryption of 128 bytes of data in CBC mode. For each cipher and each platform, the results of the implementation with the best performance indicator according to Equation (1) are shown, whereby different weights were assigned to the three metrics. Both the memory consumption and code size have a weight of 1, while the execution time has a weight of 2, which means the execution time is considered more important than the other two metrics. The Figure-of-Merit (FOM) is based on the performance indicators on all three platforms (the smaller the FOM value, the better the implementations of a cipher).

Cipher			**AVR**			**MSP**			**ARM**			
	Block [b]	Key [b]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	FOM
Chaskey	128	128	1328*	229*	**20622***	900*	**222***	**16674***	472	240	**9313**	5.4
Chaskey-LTS	128	128	1328*	229*	33102*	904*	**222***	25394*	576*	**228***	11076*	6.5
Speck	64	96	**966***	294*	39875*	664*	290*	29611*	492	308	15427	7.5
Speck	64	128	1112*	302*	41103*	**592***	298*	31832*	**444**	308	16505	7.8
Simon	64	96	1084*	363*	63649*	758*	362*	47266*	600	376	23056	10.7
Simon	64	128	1122*	375*	66613*	780*	374*	49328*	560	392	23930	11.0
LEA	128	128	1684*	631*	61020*	1154*	630*	46374*	696	644	16192	11.5
RECTANGLE	64	80	1152*	352*	66722*	832*	400*	44050*	664*	426*	35286*	12.4
RECTANGLE	64	128	1118*	353*	64813*	846*	406*	44384*	660*	432*	36121*	12.5
SPARX	64	128	1426*	392*	61955*	986*	394*	36265*	1200*	424*	40887*	13.4
SPARX	128	128	1736*	753*	83663*	1710*	758*	46640*	2290*	784*	53109*	19.6
AES	128	128	3010*	408*	58246*	2684*	408*	86506*	3080*	452*	73579*	23.6
HIGHT	64	128	1414*	333*	94557*	1258*	330*	120215*	1444*	380*	90385*	25.1
Fantomas	128	128	5892	267	111677	4164	234	56788	4604	308	70142	26.3
Robin	128	128	4944	271	146149	3170	238	76588	3572	1312	74665	28.5
Robin⋆	128	128	5076	271	157205	3312	238	88804	3724	1316	85247	31.1
RC5-20	64	128	3706	368	252368	1240	378	386026	624	376	36473	37.0
PRIDE	64	128	3384	373	111155	2918	380	226135	2240	452	130017	38.8
RoadRunneR	64	80	2504	330	144071	3088	338	235317	2788	418	119537	39.2
RoadRunneR	64	128	2316	**209**	125635	2952	362	218909	2504	448	140664	39.8
LBlock	64	80	2954	494	183324	1632	324	263778	2204	574	140647	43.7
PRESENT	64	80	2160*	448*	245232*	1838*	450*	201549*	2528*	502*	270464*	59.3
PRINCE	64	128	5358	374	243396	4174	240	357423	4372	504	201136	62.3
TWINE	64	80	4236	646	297265	3796	564	387562	2456	474	255450	70.8
Piccolo	64	80	1992	314	407269	1354	310	324221	1596	406	294478	71.9
LED	64	80	5156	574	2221555	7004	252	2065695	3696	654	594453	264.8

* Results for Assembly implementations.

ciphers based on the FOM score are Chaskey, Speck, and Simon; the FOM value of these three ciphers is less than half of the FOM value of the AES. Note that the FOM value takes into account all three metrics (i.e. execution time, RAM footprint, and code size) and does so across three platforms (i.e. AVR, MSP, and ARM). Of course, when looking at execution time, RAM requirements, or code size individually, or when looking at AVR, MSP, or ARM individually, the specific rankings can differ significantly from the overall ranking based on the FOM score. Furthermore, it should be noted that up to 24 different implementations of one and the same cipher exist, which are based on different optimization strategies. In particular, when comparing different C implementations, it can (and usually does) happen that they perform differently on the three platforms. It may also happen that one and the same cipher is slower on the 16-bit MSP platform than on 8-bit AVR (e.g. HIGHT, AES, RC5), which is not a mistake but simply due to considering RAM footprint and code size equally important as execution time. Another interesting observation is that small differences in the key length (e.g. 32 bits in the case of Simon and Speck, or even 48 bits for RECTANGLE and RoadRunneR) have only a marginal impact on the FOM value.

When having a closer look at the results on AVR, it turns out that the top-ranked algorithms are quite similar in terms of RAM footprint, which means the overall rank is primarily determined by execution time and code size. Speck has roughly twice the execution time of Chaskey, while Simon carries a performance penalty by a factor of approximately three. A somewhat surprising result is that the AES beats Simon on AVR, but its high performance comes at the expense of a rather large code size. Also LEA and SPARX are a bit faster than Simon when comparing the versions with 64-bit blocks and 128-bit keys. All other ciphers are at least three time slower than Chaskey. The situation is similar on MSP in the sense that Chaskey is

Table 4. Results for Scenario 2: Encryption of 128 bits of data (pre-computed round keys). For each cipher and each platform, the results of the implementation with the best performance indicator according to Equation (1) are shown. The Figure-of-Merit (FOM) is based on the performance indicators on all three platforms (the smaller the FOM value, the better the implementations of a cipher).

Cipher			**AVR**			**MSP**			**ARM**			
	Block [b]	Key [b]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	FOM
Chaskey	128	128	624*	80*	**1465***	388*	70*	**1153***	**216***	76*	**524***	4.4
Chaskey-LTS	128	128	624*	80*	2265*	390*	70*	1690*	**216***	76*	648*	5.0
Speck	64	96	506*	53*	2647*	**328***	**48***	1959*	256	**56**	1003	5.1
Speck	64	128	**452***	53*	2917*	332*	**48***	2013*	276	60	972	5.2
Simon	64	96	600*	57*	4269*	460*	56*	2905*	416	64	1335	7.0
Simon	64	128	608*	57*	4445*	468*	56*	3015*	388	64	1453	7.2
LEA	128	128	906*	80*	4023*	722*	78*	2814*	520*	112*	1171*	8.0
RECTANGLE	64	128	602*	56*	4381*	480*	54*	2651*	444*	76*	2365*	8.5
RECTANGLE	64	80	606*	56*	4433*	480*	54*	2651*	444*	76*	2365*	8.5
SPARX	64	128	662*	**51***	4397*	580*	52*	2261*	654*	72*	2338*	8.7
SPARX	128	128	1184*	74*	5478*	1036*	72*	3057*	1468*	104*	2935*	13.0
RC5-20	64	128	1068	63	8812	532	60	15925	372	64	1919	14.8
AES	128	128	1246*	81*	3408*	1170*	80*	4497*	1348*	124*	4044*	14.9
HIGHT	64	128	636*	56*	6231*	636*	52*	7117*	670*	100*	5532*	15.9
Fantomas	128	128	2496	108	5919	1920	78	3602	2184	184	4550	19.6
Robin	128	128	2530	108	7813	1942	80	4913	2188	184	6250	23.0
Robin*	128	128	2580	106	8052	1980	80	5262	2272	196	6417	23.7
RoadRunneR	64	80	1420	61	7329	1536	76	13034	1900	172	7234	25.5
PRIDE	64	128	2064	91	5727	1842	68	13108	1592	148	7446	25.6
RoadRunneR	64	128	1184	59	6289	1724	74	13266	1436	164	8573	26.3
LBlock	64	80	1440	64	11183	804	58	16101	1220	284	9015	28.7
PRESENT	64	80	1294*	56*	16849*	1072*	58*	12347*	1222*	80*	17105*	38.6
PRINCE	64	128	1362	72	20060	1576	76	24246	1384	280	15165	44.0
Piccolo	64	80	1114	72	25820	784	70	20081	688	112	17965	44.2
TWINE	64	80	1528	64	21701	1922	136	23662	1180	156	15673	44.6
LED	64	80	2548	267	135061	4422	104	121850	2172	352	35891	149.2

* Results for Assembly implementations.

clearly the fastest of all 19 ciphers, followed by Speck. Simon is again on the sixth position, outperformed by RECTANGLE, LEA, as well as SPARX with 64-bit blocks. However, the results on MSP also illustrate a weakness of Chaskey, namely its rather large code size, which exceeds that of Speck by a factor of 1.6. In terms of RAM consumption, PRIDE and RoadRunneR perform very well on the MSP platform. Finally, on ARM, the winners in the performance competition are Chaskey, Speck, and LEA. In addition, these three ciphers also hold the top positions in terms of code size, which is mainly because of their extremely simple round function operating on 32-bit words. All other algorithms are both slower and larger than LEA.

Table 3 shows the results for Scenario 1 when different weights are assigned to the three metrics used to calculate the performance indicator according to Equation (1), namely when the execution time has twice the weight of RAM footprint and code size. Setting the weights in this way can make sense when a cipher is used on battery-powered devices and low energy dissipation (enabled by fast execution times) is considered more valuable than low memory requirements or small code size. In this setting, the top-3 ciphers are still the same as in Table 2, which was assembled with all three metrics having the same weight, but there are some changes in the middle of the table. For example, LEA holds now the position of RECTANGLE (and vice versa) and also HEIGHT and the AES exchanged their position.

The results for Scenario 2 ("Challenge-Response Authentication") are provided in Table 4, whereby we measured the encryption of 128 bits of data in CTR mode using pre-computed round keys. We calculated the performance indicators according to Equation (1) using the default weights, which means all three metrics and all three platforms are considered equally important. The results are similar to that of Scenario 1 since the three top spots are held by the same ciphers in exactly the same order, i.e. Chaskey is the best overall performer and Speck is the runner-up. Simon secured the third place, even though on all three platforms some other ciphers show better execution times. However, Simon profits from its relatively small code size

Table 5. Results for Scenario 2: Encryption of 128 bits of data (pre-computed round keys). For each cipher and each platform, the results of the implementation with the best performance indicator according to Equation (1) are shown, whereby different weights were assigned to the three metrics. Both the memory consumption and code size have a weight of 2, while the execution time has a weight of 1, which means the former two metrics are considered more important than the execution time. The Figure-of-Merit (FOM) is based on the performance indicators on all three platforms (the smaller the FOM value, the better the implementations of a cipher).

Cipher			**AVR**			**MSP**			**ARM**			
	Block [b]	Key [b]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	FOM
Chaskey	128	128	624*	80*	**1465***	388*	70*	**1153***	**184***	76*	**568***	7.1
Speck	64	96	**448***	53*	2829*	**328***	**48***	1959*	256	**56**	1003	7.3
Speck	64	128	452*	53*	2917*	332*	**48***	2013*	264	**56**	1029	7.4
Chaskey-LTS	128	128	624*	80*	2265*	390*	70*	1690*	216*	76*	648*	7.7
Simon	64	96	534*	57*	4521*	460*	56*	2905*	416	64	1335	9.8
Simon	64	128	542*	57*	4709*	468*	56*	3015*	388	64	1453	10.0
RECTANGLE	64	128	602*	56*	4381*	480*	54*	2651*	444*	76*	2365*	11.5
RECTANGLE	64	80	606*	56*	4433*	480*	54*	2651*	444*	76*	2365*	11.5
LEA	128	128	906*	80*	4023*	722*	78*	2814*	520*	112*	1171*	12.1
SPARX	64	128	662*	**51***	4397*	580*	52*	2261*	654*	72*	2338*	12.2
RC5-20	64	128	1068	63	8812	532	60	15925	372	64	1919	18.1
SPARX	128	128	1184*	74*	5478*	904*	80*	3273*	932*	108*	4085*	19.0
HIGHT	64	128	636*	56*	6231*	636*	52*	7117*	670*	100*	5532*	19.7
AES	128	128	1246*	81*	3408*	1170*	80*	4497*	1348*	124*	4044*	21.4
Fantomas	128	128	1712	76	9689	1412	74	5506	1412	104	6484	27.7
Robin	128	128	1712	78	12499	1406	72	7051	1424	116	7686	30.9
PRIDE	64	128	958	60	11222	1842	68	13108	1592	148	7446	33.3
RoadRunneR	64	128	1184	59	6289	756	58	18067	1436	164	8573	33.3
RoadRunneR	64	80	1420	61	7329	1536	76	13034	1900	172	7234	33.7
Robin*	128	128	1754	80	14285	1980	80	5262	1472	116	9186	34.2
LBlock	64	80	1440	64	11183	804	58	16101	616	80	11818	34.7
PRESENT	64	80	1294*	56*	16849*	1072*	58*	12347*	1222*	80*	17105*	44.2
Piccolo	64	80	1114	72	25820	784	70	20081	688	112	17965	48.8
PRINCE	64	128	1362	72	20060	1578	70	24375	1200	132	16270	51.0
TWINE	64	80	1528	64	21701	1922	136	23662	1180	156	15673	52.3
LED	64	80	2602	91	143317	4422	104	121850	2172	352	35891	164.0

* Results for Assembly implementations.

and low RAM footprint. Positions 4 to 6 are held by LEA, RECTANGLE and SPARX with FOM scores that are between 1.82 and 1.98 times worse than Chaskey's FOM score. The FOM score of all other ciphers is more than three times higher than the FOM of Chaskey.

Table 5 shows the results when the performance indicators are computed using a higher weight for the RAM footprint and code size than for execution time. In real-world implementations of challenge-response authentication (e.g. access control systems), the overall latency is often determined by the data transfers between the two parties rather than the execution time of the encryption. This is, in particular, the case for RFID systems, which support only relatively low transmission rates and are also prone to transmission errors. In such a setting, one could argue that the execution time of a lightweight cipher is not the main priority (especially since the amount of data to be encrypted is small), but rather the RAM consumption and code size. The ranking of the 19 ciphers in Table 5 is based on performance indicators that assign the RAM footprint and code size twice the weight of execution time. Besides Chaskey, Speck turns out to be very lightweight and is, thus, an excellent choice for applications where size is the primary constraint. On all three platforms, Speck has a code size of below 500 Bytes and RAM footprint of less than 60 Bytes. Also Simon is size-wise consistently good on all three platforms.

Caveats. The results of any "benchmark paper" in cryptography, including ours, always reflect the state of research at the time when it was written. However, the efficient implementation of (lightweight) ciphers is an active area of research that is likely to provide new approaches for speeding up one or more of the 19 candidates considered in this paper. The AES serves as a good example on how progress in software

optimization techniques can yield significantly more efficient implementations. Similar progress could also make one or more of our lightweight ciphers much faster than anticipated today. This is the reason why we maintain a web page [19] where up-to-date benchmarking results and cipher rankings can be found. We also note that the results of most of the hardware-oriented ciphers are based on C implementations since, at the time of writing this paper, we had optimized Assembly code only for PRESENT. Although hand-crafted Assembly code is often much more efficient than compiled C code, it seems rather unlikely that Assembly programming could bring one of the hardware-tailored ciphers close to the current top performers, unless a tremendous breakthrough in software optimization is made. Furthermore, the presented results reflect, to a certain degree, also the effort the implementers have put into optimization. We invite the cryptographic research community to send us improved implementations of the 19 lightweight ciphers we analyzed in this paper. In addition, we also welcome implementations of new ciphers.

4.3 Comparison with other Benchmarking Results

Many of the ciphers we study in this paper have already been evaluated on AVR, MSP, or ARM processors before, either separately or within some other benchmarking project. It is not easily possible to compare performance figures across various frameworks and implementations because the evaluation methodology is usually different and also the optimization efforts typically vary. The importance of a consistent evaluation framework and methodology becomes quickly evident when taking the AES counter-mode implementation for Cortex-M3 processors in [49, Section 3] as example. This implementation uses the T-table approach in combination with a careful optimization of the memory accesses and achieves, according to [49], an average execution time of 659.4 clock cycles for a single-block encryption with a 128-bit key. However, this cycle count was only reached by configuring the Cortex-M3 processor to have a reduced number of wait states for memory accesses, which favors implementations using T-tables, but limits the maximum frequency the processor can be clocked with. On the other hand, our benchmarking framework operates the Cortex-M3 with the full wait states (so that it can be clocked with its maximum frequency) and reports an execution time of 1641 clock cycles for this T-table implementation. In addition, it must be taken into account that using T-tables entails a large memory footprint, which worsens the FOM score. This also explains why an implementation using only Sbox lookups can reach a better FOM score than the T-table approach, despite the fact that T-tables have the potential to reduce the execution time by a factor of more than two.

The most notable differences between our benchmarks and previous implementation results obtained on AVR, MSP, and ARM are as follows. The MSP implementations of LBlock, Piccolo, and Twine developed as part of the BLOC project [16] are a bit worse than ours, whereas the AES, HIGHT, and PRESENT are much slower. On the other hand, the AVR Assembly implementations of PRESENT and the AES from the ECRYPT project [27] are slightly slower than our Assembly versions, while our implementation of HIGHT is twice as fast as the Assembly implementation from [26] and 10 times faster than that from [27].

5 Conclusions

We presented a benchmarking framework for fair and consistent evaluation of lightweight block ciphers on three widely-used microcontroller platforms for IoT devices, namely 8-bit AVR, 16-bit MSP430, as well as 32-bit ARM Cortex-M3. The framework is able to extract three metrics of interest (execution time, RAM footprint, and binary code size) in a highly automated fashion and supports both cycle-accurate instruction set simulators and development boards. Furthermore, we introduced two usage scenarios for the evaluation of block ciphers that accomplish common IoT security services by utilizing the basic cipher operations. The framework allows one to aggregate the three extracted metrics on the three platforms into a single figure of merit according to which a set of ciphers can be ranked. With the help of this framework, we evaluated a total of 19 lightweight block ciphers using a code base consisting of over 250 different implementations altogether (including carefully-optimized Assembly implementations for nine of the 19 ciphers). Our results show that state-of-the-art ARX and ARX-like designs are not only very fast, but also extremely small in terms of RAM footprint and code size. The overall winner of our triathlon competition, based on the FOM metric, is Chaskey, closely followed by Speck. Both perform consistently well in the two usage scenarios and on all three platforms, which makes them strong candidates for a lightweight cipher to secure the IoT. Also Simon, LEA, RECTANGLE, and SPARX achieved very good results with FOM values below 10.0 when execution time, RAM footprint, and code size are considered equally important.

The FOM scores we used to rank the 19 lightweight block ciphers are solely based on efficiency metrics and do not take any (cryptanalytic) security aspects into account. In this context, it should be noted that

neither Chaskey nor versions of Speck operating on more than 32 bits provide provable security against linear or differential cryptanalysis. Also related to security is our observation that the key size has only a marginal impact on the overall efficiency of modern lightweight ciphers. In particular, the results for Simon and Speck indicate a gain in the FOM metric by a few per cent when the key size is reduced from 128 bits to 96 bits, which can hardly justify the corresponding loss of security.

The provided results can assist IoT security engineers when choosing a lightweight cipher to match the requirements of the target application and the constraints of the target device(s). Furthermore, the results are relevant for designers of ciphers as they allow them to infer some links between basic design decisions and the resulting performance and size figures when the cipher is implemented in software and executed on microcontrollers. In particular, we recommend cipher designers to focus on simple round functions that use as few operations as possible and reach a good security level after several iterations. Among the most efficient operations are the bitwise logical operations and modular addition/subtraction. The cost of rotations depends on both the features of the target architecture and the rotation amount. One should use rotations by some carefully-chosen value (e.g. 7, 8, 9, 15, or 16 bits for a 32-bit word) to reduce the execution time and code size on platforms that support only rotations by one bit at a time. To get the best performance across architectures with different word sizes, the cipher's word size should match the largest register size available on the considered architectures. In this way, the content of the registers is efficiently used on the platforms with the largest word size, while the performance on architectures with a smaller register size is not affected. The efficient operations mentioned above do not require memory accesses, provided that the cipher's state can be kept in the available registers. Finally, lookup tables of any size should be avoided as they increase the code size and/or RAM footprint and also require costly load instructions.

Future work may include the addition of new ciphers, integration of countermeasures against physical attacks, extending the toolsuite's capabilities to benchmark other lightweight symmetric primitives (stream ciphers, hash functions, authenticated encryption algorithms) and the support of additional processors.

6 Acknowledgements

We thank all contributors listed at `http://www.cryptolux.org/index.php/FELICS_Contributors` for the submitted implementations and their support for a fair evaluation of lightweight block ciphers. Daniel Dinu and Léo Perrin were supported by the CORE project ACRYPT (ID C12-15-4009992), funded by the Fonds National de la Recherche (FNR) Luxembourg.

References

1. M. R. Albrecht, B. Driessen, E. B. Kavun, G. Leander, C. Paar, and T. Yalçin. Block ciphers – Focus on the linear layer (feat. PRIDE). In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology — CRYPTO 2014*, volume 8616 of *Lecture Notes in Computer Science*, pages 57–76. Springer Verlag, 2014.
2. Arduino. Arduino Due. Specification, available online at `http://arduino.cc/en/Main/arduinoBoardDue`, 2015.
3. ARM Limited. An Introduction to the ARM Cortex-M3 Processor. White paper, available for download at `http://www.arm.com/ja/files/pdf/IntroToCortex-M3.pdf`, 2006.
4. Atmel Corporation. 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash: ATmega128, ATmega128L. Datasheet, available for download at `http://www.atmel.com/images/doc2467.pdf`, 2008.
5. L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, Oct. 2010.
6. A. Baysal and S. Sahin. RoadRunneR: A small and fast bitslice block cipher for low cost 8-bit processors. In T. Güneysu, G. Leander, and A. Moradi, editors, *Lightweight Cryptography for Security and Privacy — LightSec 2015*, volume 9542 of *Lecture Notes in Computer Science*, pages 58–76. Springer Verlag, 2016.
7. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013.
8. D. Beer. MSPDebug: Debugging tool for MSP430 MCUs. Available online at `http://mspdebug.sourceforge.net`, 2015.
9. D. J. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. Available online at `http://bench.cr.yp.to`, Feb. 2015.
10. A. Biryukov and E. Kushilevitz. Improved cryptanalysis of RC5. In K. Nyberg, editor, *Advances in Cryptology — EUROCRYPT '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 85–99. Springer Verlag, 1998.
11. C. Blondeau and K. Nyberg. Links between truncated differential and multidimensional linear properties of block ciphers and underlying attack complexities. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology — EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 165–182. Springer Verlag, 2014.

12. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. H. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems — CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer Verlag, 2007.
13. J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin. PRINCE – A low-latency block cipher for pervasive computing applications. In X. Wang and K. Sako, editors, *Advances in Cryptology — ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer Verlag, 2012.
14. C. Boura, M. Naya-Plasencia, and V. Suder. Scrutinizing and improving impossible differential attacks: Applications to CLEFIA, Camellia, LBlock and Simon. In P. Sarkar and T. Iwata, editors, *Advances in Cryptology — ASIACRYPT 2014*, volume 8873 of *Lecture Notes in Computer Science*, pages 179–199. Springer Verlag, 2014.
15. A. Canteaut, T. Fuhr, H. Gilbert, M. Naya-Plasencia, and J.-R. Reinhard. Multiple differential cryptanalysis of round-reduced PRINCE. In C. Cid and C. Rechberger, editors, *Fast Software Encryption — FSE 2014*, volume 8540 of *Lecture Notes in Computer Science*, pages 591–610. Springer Verlag, 2015.
16. M. Cazorla, S. Gourgeon, K. Marquet, and M. Minier. Implementations of lightweight block ciphers on a WSN430 sensor. Available online at `http://bloc.project.citi-lab.fr/library.html`, 2015.
17. M. Cazorla, K. Marquet, and M. Minier. Survey and benchmark of lightweight block ciphers for wireless sensor networks. In P. Samarati, editor, *Proceedings of the 10th International Conference on Security and Cryptography (SECRYPT 2013)*, pages 543–548. SciTePress, 2013.
18. H. Chen and X. Wang. Improved linear hull attack on round-reduced Simon with dynamic key-guessing techniques. Cryptology ePrint Archive, Report 2015/666, 2015.
19. CryptoLUX Team. FELICS: Fair Evaluation of Lightweight Cryptographic Systems. Available online at `http://www.cryptolux.org/index.php/FELICS`, 2016.
20. J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen. Nessie proposal: NOEKEON. Specification, available for download at `http://gro.noekeon.org/Noekeon-spec.pdf`, 2000.
21. J. Daemen and V. Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer Verlag, 2002.
22. P. Derbez and P.-A. Fouque. Exhausting Demirci-Selçuk meet-in-the-middle attacks against reduced-round AES. In S. Moriai, editor, *Fast Software Encryption — FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 541–560. Springer Verlag, 2013.
23. D. Dinu, L. Perrin, A. Udovenko, V. Velichkov, J. Großschädl, and A. Biryukov. Design strategies for ARX with provable bounds: Sparx and LAX. In J. H. Cheon and T. Takagi, editors, *Advances in Cryptology — ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 484–513. Springer Verlag, 2016.
24. I. Dinur, O. Dunkelman, N. Keller, and A. Shamir. Key recovery attacks on 3-round Even-Mansour, 8-step LED-128, and full AES^2. In K. Sako and P. Sarkar, editors, *Advances in Cryptology — ASIACRYPT 2013*, volume 8269 of *Lecture Notes in Computer Science*, pages 337–356. Springer Verlag, 2013.
25. T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni, F.-X. Standaert, and L. van Oldeneel tot Oldenzeel. Compact implementation and performance evaluation of block ciphers in ATtiny devices. In A. Mitrokotsa and S. Vaudenay, editors, *Progress in Cryptology — AFRICACRYPT 2012*, volume 7374 of *Lecture Notes in Computer Science*, pages 172–187. Springer Verlag, 2012.
26. T. Eisenbarth, S. S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6):522–533, Nov. 2007.
27. European Network of Excellence in Cryptology (ECRYPT II). Implementations of Low Cost Block Ciphers in Atmel AVR Devices. Available online at `http://perso.uclouvain.be/fstandae/lightweight_ciphers`, 2015.
28. D. Evans. The Internet of Things: How the Next Evolution of the Internet is Changing Everything. Cisco IBSG white paper, available for download at `http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, Apr. 2011.
29. M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 357–370. Springer Verlag, 2004.
30. V. D. Gligor. Light-weight cryptography – How light is light? Keynote presentation at the Information Security Summer School, Florida State University. Slide deck available online at `http://www.sait.fsu.edu/conferences/2005/is3/resources/slides/gligorv-cryptolite.ppt`, May 2005.
31. V. Grosso, G. Leurent, F.-X. Standaert, and K. Varici. LS-designs: Bitslice encryption for efficient masked software implementations. In C. Cid and C. Rechberger, editors, *Fast Software Encryption — FSE 2014*, volume 8540 of *Lecture Notes in Computer Science*, pages 18–37. Springer Verlag, 2015.
32. J. Guo, T. Peyrin, A. Poschmann, and M. J. Robshaw. The LED block cipher. In *Cryptographic Hardware and Embedded Systems — CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer Verlag, 2011.
33. B. Han, H. Lee, H. Jeong, and Y. Won. The HIGHT Encryption Algorithm. Internet Engineering Task Force, Network Working Group, Internet draft draft-kisa-hight-00 (work in progress), Dec. 2011.
34. D. Hong, J.-K. Lee, D.-C. Kim, D. Kwon, K. H. Ryu, and D. Lee. LEA: A 128-bit block cipher for fast encryption on common processors. In Y. Kim, H. Lee, and A. Perrig, editors, *Information Security Applications — WISA 2013*, volume 8267 of *Lecture Notes in Computer Science*, pages 3–27. Springer Verlag, 2013.

35. D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee. HIGHT: A new block cipher suitable for low-resource device. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems — CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 46–59. Springer Verlag, 2006.
36. IEEE Standards Association. IEEE 802.15.4-2015 – IEEE Standard for Low-Rate Wireless Networks. Available online at `http://standards.ieee.org/findstds/standard/802.15.4-2015.html`, 2015.
37. A. Journault, F.-X. Standaert, and K. Varici. Improving the security and efficiency of block ciphers based on LS-designs. *Designs, Codes and Cryptography*, 82(1–2):495–509, Jan. 2017.
38. K. Khoo, T. Peyrin, A. Y. Poschmann, and H. Yap. FOAM: Searching for hardware-optimal SPN structures and components with a fair comparison. In L. Batina and M. J. Robshaw, editors, *Cryptographic Hardware and Embedded Systems — CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 433–450. Springer Verlag, 2014.
39. G. Leander, B. Minaud, and S. Rønjom. A generic approach to invariant subspace attacks: Cryptanalysis of Robin, iSCREAM and Zorro. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology — EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 254–283. Springer Verlag, 2015.
40. G. Leurent. Improved differential-linear cryptanalysis of 7-round Chaskey with partitioning. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology — EUROCRYPT 2016*, volume 9665 of *Lecture Notes in Computer Science*, pages 344–371. Springer Verlag, 2016.
41. F. Mendel, V. Rijmen, D. Toz, and K. Varici. Differential analysis of the LED block cipher. In X. Wang and K. Sako, editors, *Advances in Cryptology — ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 190–207. Springer Verlag, 2012.
42. N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede. Chaskey: An efficient MAC algorithm for 32-bit microcontrollers. In A. Joux and A. M. Youssef, editors, *Selected Areas in Cryptography — SAC 2014*, volume 8781 of *Lecture Notes in Computer Science*, pages 306–323. Springer Verlag, 2014.
43. National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). FIPS Publication 197, available for download at `http://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf`, 2001.
44. National Institute of Standards and Technology (NIST). Lightweight Cryptography Project. Available online at `http://csrc.nist.gov/projects/lightweight-cryptography`, 2016.
45. National Institute of Standards and Technology (NIST). SHA-3 Project. Available online at `http://csrc.nist.gov/projects/hash-functions/sha-3-project`, 2016.
46. O. Özen, K. Varici, C. Tezcan, and Ç. Kocair. Lightweight block ciphers revisited: Cryptanalysis of reduced round PRESENT and HIGHT. In C. Boyd and J. G. Nieto, editors, *Information Security and Privacy — ACISP 2009*, volume 5594 of *Lecture Notes in Computer Science*, pages 90–107. Springer Verlag, 2009.
47. A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. SPINS: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, Sept. 2002.
48. R. L. Rivest. The RC5 encryption algorithm. In B. Preneel, editor, *Fast Software Encryption — FSE '94*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer Verlag, 1995.
49. P. Schwabe and K. Stoffelen. All the AES you need on Cortex-M3 and M4. In R. M. Avanzi and H. M. Heys, editors, *Selected Areas in Cryptography — SAC 2016*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer Verlag, 2017.
50. K. Shibutani, T. Isobe, H. Hiwatari, A. Mitsuda, T. Akishita, and T. Shirai. Piccolo: An ultra-lightweight blockcipher. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems — CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 342–357. Springer Verlag, 2011.
51. L. Song, Z. Huang, and Q. Yang. Automatic differential analysis of ARX block ciphers with application to SPECK and LEA. Cryptology ePrint Archive, Report 2016/209, 2016.
52. T. Suzaki, K. Minematsu, S. Morioka, and E. Kobayashi. TWINE: A lightweight, versatile block cipher. In G. Leander and F.-X. Standaert, editors, *Proceedings of the 1st ECRYPT Workshop on Lightweight Cryptography (LC 2011)*, pages 146–169, 2011.
53. Texas Instruments. MSP430x1xxx Family User's Guide. Available for download at `http://www.ti.com/lit/ug/slau049f/slau049f.pdf`, 2006.
54. B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, pages 477–482. IEEE, 2005.
55. B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: The AVR simulation and analysis framework. Available online at `http://compilers.cs.ucla.edu/avrora`, 2005.
56. Y. Wang and W. Wu. Improved multidimensional zero-correlation linear cryptanalysis and applications to LBlock and TWINE. In W. Susilo and Y. Mu, editors, *Information Security and Privacy — ACISP 2014*, volume 8544 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 2014.
57. C. Wenzel-Benner and J. Gräf. XBX: eXternal Benchmarking eXtension for the SUPERCOP crypto benchmarking framework. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems — CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 294–305. Springer Verlag, 2010.
58. W. Wu and L. Zhang. LBlock: A lightweight block cipher. In J. López and G. Tsudik, editors, *Applied Cryptography and Network Security — ACNS 2011*, volume 6715 of *Lecture Notes in Computer Science*, pages 327–344. Springer Verlag, 2011.

59. Q. Yang, L. Hu, S. Sun, K. Qiao, L. Song, J. Shan, and X. Ma. Improved differential analysis of block cipher PRIDE. In J. López and Y. Wu, editors, *Information Security Practice and Experience — ISPEC 2015*, volume 9065 of *Lecture Notes in Computer Science*, pages 209–219. Springer Verlag, 2015.
60. Q. Yang, L. Hu, S. Sun, and L. Song. Extension of meet-in-the-middle technique for truncated differential and its application to RoadRunneR. In J. Chen, V. Piuri, C. Su, and M. Yung, editors, *Network and System Security — NSS 2016*, volume 9955 of *Lecture Notes in Computer Science*, pages 398–411. Springer Verlag, 2016.
61. W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang, and I. Verbauwhede. RECTANGLE: A bit-slice lightweight block cipher suitable for multiple platforms. *Science China Information Sciences*, 58(12):1–15, Dec. 2015.
62. ZigBee Alliance. ZigBee Wireless Standard. Available online at `http://www.zigbee.org`, 2015.

A Target Devices

8-bit AVR ATmega128 Microcontroller. The ATmega128 [4] microcontroller developed by Atmel is based on an 8-bit RISC architecture and provides 133 instructions, which are encoded to be either 16 or 32 bits wide. Most of the instructions are executed in only one or two clock cycles. The ATmega128 features a two-stage pipeline, making it possible to execute an instruction while the next instruction is fetched from program memory. In addition, it comes with a relatively large register file consisting of 32 general-purpose registers (`R0` to `R31`) of 8-bit width. Six registers can be used as three 16-bit pointers (`X`, `Y`, and `Z`) to access the data space. All 32 registers are directly connected to the Arithmetic Logic Unit (ALU). The standard ALU instructions have a two-address format, which allows them to read two 8-bit operand words from two independent registers and write the result of the operation back to one of them. Like other members of the 8-bit AVR family, the ATmega128 uses a Harvard architecture (i.e. separate memories, buses, and address spaces for program and data) to maximize performance and parallelism. The memory sub-system includes 128 kB of flash (for storing program code), 4 kB of SRAM, and 4 kB of EEPROM.

16-bit MSP430F1611 Microcontroller. The MSP430F1611 [53] is a 16-bit microcontroller from Texas Instruments that contains a RISC CPU optimized for ultra-low power consumption and various peripheral modules. A distinguishing feature of the MSP430 architecture is its minimalist instruction set comprising only 27 core instructions and 24 emulated instructions. The length of an instruction can vary between one and three 16-bit words, i.e. between two and six bytes. Depending on the instruction format, the 27 core instructions fall into three categories: double-operand instructions (which overwrite one of the two operands with the result), single-operand instructions, and jumps. The MSP430 instruction set is highly orthogonal and supports seven addressing modes for the source operand and four addressing modes for the destination operand. Depending on the used addressing modes, double-operand instructions have a latency of between one clock cycle (when source and destination operands are held in registers) and six clock cycles (operands are in RAM or flash). There are 16 registers, of which four, namely `R0` to `R3`, serve a special purpose. The von-Neumann memory system of the MSP430F1611 consists of 10 kB RAM and 48 kB flash.

32-bit ARM Cortex-M3 Microcontroller. The Cortex-M3 is a member of the ARM Cortex-M series of 32-bit microcontrollers that was specifically designed to achieve high system performance in power- and cost-sensitive embedded applications [3]. It is based on the ARMv7-M architecture and supports Thumb-2 technology, which extends the 16-bit fixed-width Thumb instruction set with some additional 32-bit ARM instructions, whereby 16-bit and 32-bit instructions can be freely intermixed. Data processing instructions have a conventional three-address format that allows the target register to be distinct from the two source operands. The first operand must always be one of the 13 general-purpose 32-bit registers, while the second operand can be a register, an immediate value, or a register with an optional shift. Many instructions can be executed conditionally, based on condition flags set by another instruction. Cortex-M3 microcontrollers incorporate a Harvard architecture (enabling simultaneous instruction fetch with data load/store) and have a three-stage pipeline with branch speculation. The specific Cortex-M3 device we use for benchmarking is an Arduino Due board equipped with an Atmel SAM3X8 that features 512 kB flash and 96 kB RAM.

B API and Implementation Requirements

To unify evaluation conditions, our framework imposes some requirements on the implementation of a block cipher. Firstly, basic operations must be performed by functions having the following C prototypes.

```
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);
```

Each of the above functions should be implemented in its own C file. If the cipher key schedule is the same for encryption and decryption then only the encryption key schedule function has to be implemented. The framework takes a common key schedule into account when computing the different metrics. Secondly, all other common code sections should be placed in separate functions to reduce the overall code size. The implementer needs to add the names of the common files to the implementation info file, which gets parsed by the framework when extracting the three metrics for the implementation. Thirdly, the implementer has to choose whether the constants used by the cipher should be stored in flash/ROM or RAM. However, this flexibility comes at the expense that the implementer has to define and use a dedicated macro to read the constant value(s). Fourthly, the block size used by the implementation must be a multiple of 64 bits.

While these requirements guarantee the same evaluation conditions for an accurate assessment of the performance of a block cipher in various different evaluation scenarios, they limit the applicability of some optimization techniques like bit-slicing. Even though bit-sliced implementations can be very fast, they have the disadvantage of high memory consumption and can only be used in non-feedback modes of operation (e.g. CTR mode). However, the performance of a cipher implementation in such (highly) specific settings does not say anything about the cipher's performance in more general usage scenarios, which is what we are mainly interested in and our framework was designed for. The benchmarking toolsuite is able to verify the compliance with the formulated requirements and to check the correctness of an implementation with the help of test vectors. Since the metrics extraction process is completely automated, the toolsuite is easy to use, even for beginners with little experience.