Triathlon of Lightweight Block Ciphers for the Internet of Things

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

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

Abstract. In this paper we introduce an open framework for the benchmarking of lightweight block ciphers on a multitude of embedded platforms. Our framework is able to evaluate execution time, RAM footprint, as well as (binary) code size, and allows a user to define a custom "figure of merit" according to which all evaluated candidates can be ranked. We used the framework to benchmark various implementations of 13 lightweight ciphers, namely AES, Fantomas, HIGHT, LBlock, LED, Piccolo, PRESENT, PRINCE, RC5, Robin, Simon, Speck, and TWINE, on three different platforms: 8-bit ATmega, 16-bit MSP430, and 32-bit ARM. Our results give new insights to the question of how well these ciphers are suited to secure the Internet of Things (IoT). The benchmarking framework provides cipher designers with a tool to compare new algorithms with the state-of-the-art and allows standardization bodies to conduct a fair and comprehensive 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") that have the ability to communicate with each other and with 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 key 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 estimates no less than 50 billion devices being connected to the Internet by the year 2020 [25], which implies that, in the near future, every person in the industrialized world will be surrounded by hundreds 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 profound 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 so as to comply with the scarce resources of typical IoT devices. Gligor defined in [26] lightweight cryptography as *cryptographic primitives, schemes and protocols tailored to extremely constrained environments such as sensor nodes or RFID tags.* A typical sensor node (e.g. the MICAz mote) is equipped with an 8-bit micro-controller (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 are applicable in the highly constrained regimes of sensor nodes and RFID tags is a challenging task since, for example, performance is conflicting with other metrics of interest such as silicon area and power consumption (when thinking of hardware implementation) or memory footprint and code size (when implementing in software). In addition, lightweight primitives should withstand all known forms of cryptanalytic attacks (e.g. linear and differential cryptanalysis in the context of secret-key primitives) since lightweight cryptography is not meant to be "weak" cryptography, i.e. a lightweight cryptographic primitive should not be the weakest link in the security of a system [26].

* The authors are supported by the CORE project ACRYPT funded by the FNR Luxembourg

In this paper we present a survey of lightweight block ciphers along with software benchmarking results we obtained on 8, 16, and 32-bit processors. We consider three metrics of interest: execution time, run-time memory (i.e. RAM) requirements, and binary code size. To ensure a fair and consistent evaluation, we developed a benchmarking toolsuite that we plan to make available to the cryptographic research community following the spirit of the well-known and widely-used eBACS [9] system. Our benchmarking tool is "open" in various aspects; first, it will be 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 prototyping 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 M processor. We use GCC for all 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 consumption. However, since our framework supports prototyping boards, it can be easily extended to collect more detailed and precise energy figures by measuring the power consumption.

Our benchmarking toolsuite accepts implementations written in ANSI C, whereby performance-critical code sections can be accelerated with inlined Assembly code 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 performance optimized implementations played an important role in the evaluation of candidates for cryptographic standards, such as the AES [40] and SHA-3 [37], and this will also be the case for potential future standardization activities in the area of lightweight cryptography for the IoT [36]. However, any implementation of a cipher written in Assembly language is processor-dependent and, therefore, not portable. At the opposite 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 high-speed Assembly implementations is out of dispute, we argue in this paper that there is also a need to benchmark portable C implementations of lightweight ciphers. Our argument is twofold and based on the requirements 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 operating systems. Supporting all these processors through optimized Assembly code is tedious and error-prone since, for each processor architecture, a separate code base needs to be written, tested, debugged, and maintained. In the light of ever-increasing time-to-market pressure, many cryptographic engineers value the portability of C code higher than the performance of Assembly code[1]. Even though we mentioned before that, for some ciphers, Assembly optimizations yield a significant speed-up, it should be noted that there are also many ciphers for which this does not hold. In particular, if a cipher mainly performs single-word arithmetic/logical operations, then optimizing compilers generally produce high-quality code that comes very close to a hand-crafted Assembly implementation. In such a case, Assembly programming would introduce a significant overhead for a relatively little gain. Our second argument in favor of portable C implementations is related to the steadily increasing research interest in lightweight ciphers with new designs being published (almost) every week. 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. Our benchmarking toolsuite allows cipher designers to 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.

[1] On the other hand, if there is only one single platform to support, as is the case with the "conventional" Internet where the Intel architecture enjoys an almost monopoly, it makes of course sense to use a performance-optimized Assembly implementation.

We report detailed benchmarking results for a total of 13 lightweight block ciphers, namely AES, Fantomas, HIGHT, LBlock, LED, Piccolo, PRESENT, PRINCE, RC5, Robin, Simon, Speck, and TWINE. Our rationale behind selecting exactly the mentioned 13 ciphers is twofold; first, each of these candidates has a special property or feature that makes it interesting for IoT applications. Second, they cover a wide range of different design strategies and approaches. Our evaluation considers two applications scenarios or use cases; the first considers 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 of these application scenarios, we implemented at least two versions of most of the 13 ciphers, a memory-optimized 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 incorporates certain optimizations that increase code size and/or memory footprint (e.g. partial loop unrolling, use of small look-up tables) with the goal of improving performance. Almost all our implementations are written in C language to ensure portability across a wide range of 8, 16, and 32-bit platforms[2]. We put a similar effort into optimizing the implementations of all ciphers to ensure a consistent and fair evaluation. Even though our C implementations do not achieve record-setting execution times, the benchmarks we report in this paper are still practically relevant. On the one hand, they will be useful for cryptographic engineers who value portability higher than performance. For example, our results can assist them in choosing the block cipher that best meets the requirements of a target application with respect to execution time, RAM footprint and code size. On the other hand, also cipher designers will profit from our benchmarking results because they allow them to assess how a new design candidate compares with the state-of-the-art.

Related Work. Several similar projects evaluated the performance of lightweight block ciphers, but none of them addressed the specific constraints imposed by the context of IoT. We chose to develop a new framework instead of contributing to an existing one, because none of the existing projects provides a fair and flexible benchmarking framework capable of extracting comprehensive results using accurate measurements for the target devices commonly used in IoT context. Nevertheless, understanding the strengths and weaknesses of previous benchmarking frameworks helped us to design a more flexible and powerful framework for evaluation of lightweight ciphers on different embedded devices commonly used in the IoT context.

The authors of [17] analysed 16 lightweight block ciphers on the MSP430 16-bit microcontroller. The provided C library [16] shows that the project does not use a common interface for evaluating all the ciphers and it can not easily accommodate new devices. Inspecting the benchmarking code we discovered that the RAM requirement metric is wrongly computed, because the implementers assume that the `unsigned int` data type takes one byte on the target microcontroller instead of two. Although this is the biggest collection of lightweight ciphers implementations available, some of the implemented ciphers do not verify the test vectors provided in the cipher specifications. We created a patch that fixes the identified issues and submitted it to the authors of the project, but the patch is not yet applied to the public repository since the project is not active for more than ten months. For the ciphers considered both in our paper and in [17], our results on the same platform are on average three times better.

During the ECRYPT II project, a survey paper [22] concerning the performance evaluation of 12 low-cost block ciphers on AVR ATtiny45 device was published. The set of analysed ciphers includes lightweight ciphers designed until the paper publication and thus it does not contain recent designs. The authors described the methodology used and the requirements formulated to ensure a fair comparison of the lightweight block ciphers. Although the assembly implementations are available [23], there is no framework provided that can help users to asses the performance of new designs in the same conditions. The assembly implementation results of this survey on AVR ATtiny45 are on par with our assembly implementation results on AVR ATmega128, while the assembly implementation in [22] is five times slower than our C implementation.

The XBX extension [47] to SUPERCOP [9] allowed benchmarking hash functions on embedded devices, adding at the same time two new metrics, namely RAM footprint and ROM consumption. The framework is not maintained any more, but still worth mentioning because of the consistent evaluation across several embedded devices and the importance of the results in the context of the SHA-3 competition [37].

[2] The AES and PRESENT are exceptional cases in the sense that our C implementations are significantly slower than hand-optimized Assembly code. Therefore, we include the results of Assembly implementations for these two ciphers (see Subsection 6.3 for further discussion).

Our Contributions. Firstly, 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 full source code of our framework available under GPL to facilitate the establishment of a completely free and open benchmarking environment.

Secondly, we survey a total of 13 lightweight block ciphers and study, in particular, their suitability for software implementation on resource-restricted processors. This set of ciphers covers a wide range of different design principles and includes some recent proposals with very interesting properties, e.g. Simon/Speck and Robin/Fantomas. We collected between two and up to 24 implementations of each of cipher to account for different trade-offs between execution time, RAM footprint and code size. In total, our repository includes over 80 implementations, of which we developed roughly two third from scratch and the rest we took over (and slightly modified) from other open-source projects or directly from the designers. The source code of all our implementations is available under GPL.

Thirdly, we report detailed performance, RAM footprint and code size figures of the 13 block ciphers, which we generated with the help of our benchmarking toolsuite. We also define two typical usage scenarios that resemble security-related operations commonly performed by real-world IoT devices. The results we obtained shed new light on the relative performance of lightweight block ciphers because (1) some of our implementations are significantly faster than that of previous survey and benchmarking efforts, and (2) 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 [18] with the most recent 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 ciphers can be assembled. The FOM metric can assign different weights to 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 Usage Scenarios & Target Devices

We chose to evaluate the studied block ciphers in two usage scenarios that cover the two main security requirements of the IoT: communication confidentiality and entity authentication. The target devices are three widely used microcontrollers having low power consumption. We selected these devices because they match the IoT resource constraints and are good representatives of the most-widely used 8, 16 and 32-bit platforms.

Scenario 1 – Communication Protocol This scenario covers the need for secure communication in sensor networks and between IoT devices. It assumes that the sensitive data is encrypted and decrypted using a lightweight block cipher in CBC mode of operation. Considering the limitations of communication protocols in sensor networks described in IEEE 802.15.4 [31] and ZigBee [50] standards, the data length exchanged in a single transmission by IoT constrained devices is 128 bytes. Because the message length is fixed to 128 bytes, we do not consider a padding scheme since this introduces unnecessary overhead. The IoT device has the cipher master key stored in RAM and the round keys are computed using the key schedule and then stored in RAM for later use in the encryption process. The data that has to be sent as well as the initialization vector are also stored in the device's RAM. Encryption is performed in place to reduce the RAM consumption. The key schedule does not modify the master key since it may be used later.

Scenario 2 – Challenge-Handshake Authentication Protocol Challenge-handshake authentication covers the need of authentication in the IoT. The scenario assumes an authentication protocol, where the block cipher is used in CTR mode to encrypt 128 bits of data. The device has the cipher round keys stored in Flash memory and there is no master key stored into the device and consequently no key schedule is required. The data that has to be encrypted is stored in RAM, as well as the counter value. To reduce the RAM usage, the encryption process is done in place. This scenario is suitable for very constrained environments where binary code size and RAM usage have to be extremely low, while the execution time should be fast enough to prevent depleting the device's battery.

Target Devices The three microcontrollers selected for our evaluation are the 8-bit AVR ATmega128 [4], the 16-bit TI MSP430F1611 [32] and the Arduino Due [2] based on 32-bit ARM Cortex-M3 [3]. They use RISC microprocessors clocked at 16, 8, respectively 84 MHz. For a brief description of each microcontroller see Appendix A.

3 Benchmarking Framework

Most papers introducing a new cipher report performance evaluation on different platforms and usually in different conditions. The results obtained on different devices and in different measurement conditions are then used to compare the new cipher with previous ones. The conclusions are not accurate and do not inspire confidence because it is hard to correctly evaluate different ciphers if comparative implementations are not available. Our benchmarking framework is motivated by the need for a unified evaluation of lightweight block ciphers' performances.

We introduce the tool used to collect performance metrics for lightweight ciphers on three different devices: 8-bit AVR, 16-bit MSP and 32-bit ARM. To increase the level of confidence and transparency in our results, the framework is available for free together with more than 80 implementations of 13 lightweight block ciphers.

We strived to make our framework both easy to use and flexible, hence our choice to benchmark C and assembly implementations. Considering that almost all ciphers' reference implementations are written in C, the cipher designer simply has to adapt their reference implementation to the framework requirements to be able to evaluate the new cipher. The C language is widespread and easy to cross compile for the selected architectures. Therefore, a lightweight block cipher's performance can be evaluated on three different IoT platforms with limited efforts. If top performance figures are more important than portability, then highly optimised assembly implementations for the supported target devices can be evaluated using the same interface.

To ensure a fair evaluation, we formulated a set of constraints that each implementation should follow. The detailed description of the framework requirements is given in Appendix B.

The benchmarking framework is able to extract three primary metrics: code size, RAM consumption and execution time. We consider these metrics because they describe the cipher's characteristics with respect to the IoT device requirements and they can not be inferred from other metrics. We do not consider derived or secondary metrics such as energy consumption, power consumption, etc. Those metrics are closely related to the basic metrics and thus would be redundant. For example, the energy consumption can be computed from the device's energy model and the execution time.

It is difficult to optimize the three metrics simultaneously. Thus, we tried different trade-offs in order to better understand the link between each cipher construction and the performance figures. We introduce the Figure-of-Merit (FOM), which is a weighted sum of each cipher's performance across the three metrics. The results extracted by the framework are very detailed and will help embedded devices programmers to chose between different implementation strategies depending on their particular constraints.

3.1 Code Size

The code size is measured in bytes and corresponds to the program footprint which is stored in the Flash memory of the target device. The code size for each cipher implementation is computed using the `size` tool on object files generated by the compiler. The tool lists the section sizes for each analyzed binary file. To get the code size requirement we add the value of the `text` and `data` sections of the relevant object files. The `text` section of a binary file contains the code, while the `data` section contains global initialized variables. The content of the `data` section is loaded from Flash into RAM at execution time. Since our framework forbids the use of global uninitialized variables in cipher implementation, we do not consider the `bss` section of the binary file, which gives the code size for global uninitialized variables.

The common code parts are considered just once when computing the code size for operations that use the common code several times to encourage code reuse. For example, it can help implementers to decide if they should implement only encryption or encryption and decryption operations. The measurements do not consider the `main` function's code size, where all the cipher operations are put together. This value is the same for all ciphers and is not relevant for the studied scenarios.

3.2 RAM

The RAM consumption is divided into stack consumption and data consumption. The size of the data stored in RAM is computed using the implementation information file and the `size` tool. It includes scenario specific RAM data such as data to encrypt, keys, round keys or initialization vectors. The stack consumption is measured using `gdb`. At the beginning of each of the measured operations, immediately after the function call for that operation the stack is filled with a memory pattern. At the end of the function execution, the values in the memory are compared with the memory pattern and the number of modified bytes gives the stack consumption. There are no arguments saved on the stack that have to be counted, because the measured functions do not use value arguments. The return address is not considered as stack consumption since it is insignificant and the same for all ciphers on a given target device.

3.3 Execution Time

The execution time is expressed in number of processor cycles spent executing a set of instructions. The number of processor cycles is given by the number of cycles of the processor's clock. The metric is extracted for the four basic operations performed by a block cipher. To measure the execution time on AVR we used the cycle accurate simulator `Avrora` [45,44]. For MSP we used the cycle accurate simulator `MSPDebug` [8]. To extract the execution time metric on ARM microprocessor we inserted additional C code for reading the system timer number of ticks at the beginning and at the end of the measured operations. The difference between these values gives the number of cycles required for the measured operation. To extract the values we used the Arduino Due [2] board based on ARM Cortex-M3 [3] microprocessor. The extracted values for ARM may vary depending on how the C instructions are translated into assembly instructions by the compiler in different contexts and how the data types are aligned in memory. This is the reason why we get different cycle count values in different usage scenarios for the same function.

4 Analysed Ciphers

Since our aim is to understand the link between the cipher construction and the performance figures on the selected devices in the IoT context, we selected ciphers representing a large variety of design decisions from the two large families of Substitution-Permutation Networks (SPN) and Feistel Networks (FN).

The AES is the canonical example of an SPN, however other designs for the S-box and the linear layer are of course possible (ex. PRESENT, Robin and Fantomas). The overall structure of the cipher can also vary while still maintaining a round function consisting in an S-box layer and a linear layer: LED adds key material every 4 rounds only while PRINCE implements a unique property called α-reflection.

Feistel Networks can be designed using a small SPN as the Feistel function, as in LBlock or Piccolo, or using simple arithmetic and logical operations as in Simon and in ARX designs like HIGHT, SPECK and RC5. These operation may be data-dependent as is the case 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 in a simple rotation (HIGHT) or in a dedicated permutation optimizing diffusion (TWINE, Piccolo). A high number of branches allows the use of very simple Feistel functions as in TWINE and HIGHT.

We chose to evaluate a wide variety of lightweight ciphers, both hardware and software oriented, because in the IoT context it is expected that the hardware and software devices to communicate to each other. Although we included a similar number of hardware and software oriented designs, we notice that the trend is moving from hardware only ciphers (Piccolo, PRINCE) to software friendly designs (Simon, Speck, Fantomas, Robin).

Block sizes of 64-bits are used when available, otherwise 128-bits are used. We use the cipher version with key length that is greater than or equal to 80 bits, which is considered sufficient in the context of the IoT. We provide a brief description of each cipher and refer the reader to the original papers for more details.

Table 1. Studied ciphers. Block, key and round keys sizes are expressed in bits. 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 keys size	Rounds	Security level	Type	Target
AES	1998	128	128	1408	10	0.70	SPN	SW, HW
Fantomas	2014	128	128	0	12	NA	SPN	SW
HIGHT	2006	64	128	1088	32	0.69	Feistel	HW
LBlock	2011	64	80	1024	32	0.72	Feistel	HW, SW
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
PRINCE	2012	64	128	192	12	0.83	SPN	HW
RC5*	1994	64	128	1344	20	0.90	Feistel	SW
Robin	2014	128	128	0	16	NA	SPN	SW
Simon	2013	64	96	1344	42	0.67	Feistel	HW, SW
Speck	2013	64	96	832	26	0.58	Feistel	SW, HW
TWINE	2011	64	80	1152	36	0.64	Feistel	HW, SW

* We use RC5 with increased number of rounds.

AES has been standardized by the American NIST and is widely used. It has a SPN structure with an internal state of 128-bits represented as 4×4 byte matrix. The `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` operations are applied to the cipher state [40,19]. The best single-key cryptanalysis of AES-128 is a meet-in-the-middle attack on 7 rounds out of 10 [20].

Fantomas is a 128-bits block cipher similar to Robin. It is a LS-design, meaning that the linear layer consists in the parallel applications of so-called "L-boxes". The S-box structure simplifies the implementation of masking. There is no key-schedule: the master key is added at every round [27]. At the time of writing, there was to the best of our knowledge no attack on this very recently designed block cipher.

HIGHT is a generalized Feistel network with an ARX structure. Indeed, the Feistel functions are built using only XOR and bitwise rotations. The output of the Feistel functions is combined with the other branches using either XOR or addition modulo 2^8 [30]. A saturation attack breaks 22 out of 32 rounds of this cipher [49].

LBlock is a Feistel Network with 32 rounds. The Feistel function consists of a XOR with the round subkey, a substitution layer of 8 different S-boxes and a permutation of 8 nibbles. Furthermore, the content of one of the branches is rotated by 8 bits in each round. The design trade-offs between security and performance led not only to hardware efficiency but also software efficiency [48]. The best cryptanalysis of this primitive is an impossible differential attack on 23 out of 32 rounds [14].

LED is an AES-based cipher aiming at very compact hardware implementation while maintaining reasonable performance in software. The main characteristic of the cipher is the absence of the key schedule, the round keys being replaced with a part of the master key [28]. To the best of our knowledge, there are no attacks on LED-80. However, there is a differential attack covering 16/32 rounds of LED-64 and 24/48 rounds of LED-128 [35]. The structural attack breaking 32/48 rounds of LED-128 presented in [21] is unlikely to be adapted to attack LED-80.

Piccolo is a generalized Feistel structure with four 16-bit branches. To improve diffusion, Piccolo uses a byte permutation between rounds. The Feistel function consists of two S-box layers separated by a diffusion matrix [42]. The best attack on Piccolo-80 is a Meet-in-the-Middle attack presented by its designers in the paper introducing the cipher.

PRESENT is a SPN based block cipher with a bit oriented permutation layer. The non-linear layer is based on a single 4-bit S-box which was designed with hardware optimizations in mind [12]. A truncated differential attack on 26 out 31 of rounds of PRESENT is presented in [11].

PRINCE uses a so-called FX construction, where the first two subkeys are used as whitening keys, while the third subkey is the 64-bit key for the 12 rounds SPN called PRINCE_{core}. The cipher introduces the α-reflection property: encryption with one key corresponds to decryption with a related key [13]. The best attack on this cipher is a multiple differential attack on 10 out of 12 rounds [15].

RC5 is a Feistel network which uses data dependent rotations [41]. Though RC5 was designed before lightweight cipher design became popular, it is obviously lightweight, which is confirmed by its wide use in sensor networks. The block and key size as well as the number of rounds can be chosen freely so we study RC5-32/20/16, i.e. a version of RC5 operating on two 32 bit words, using 20 rounds (40 half-rounds) and a 16 byte key. We have chosen the number of rounds so as to have a security level of 0.90. RC5-32/12/16 can be attacked using a differential cryptanalysis [10] which can be extrapolated to 18 rounds but would require almost the full codebook (2^{64} ciphertexts).

Robin is a 128-bits block cipher similar to Fantomas but, for example, its "L-boxes" are involutions. The look-up table-based diffusion layers and the structure of the S-boxes makes the family ciphers good candidates for Boolean masking in bitslice software implementations [27]. There exists a set of weak keys of density 2^{-32} for this cipher which, if used, leads to attack on the full primitive [34].

Simon uses a Feistel structure with a simple round function which uses bitwise XOR, bitwise AND and left circular shifts. It is optimized for performance in hardware implementations, but achieves good results in software also [6]. A differential attack on 28 out of 42 rounds of Simon-64/96 is presented in [38].

Speck is designed to provide excellent results in both hardware and software, but is optimized for software implementation on microcontrollers. 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 [6]. The best attack against Speck-64/96 is a differential attack on 15 out of 26 rounds [1].

TWINE is a generalized Feistel Network with 16 branches. The Feistel function consists simply in a key addition and the application of a 4-bit S-box. The linear layer is a nibble permutation with much higher diffusion than a nibble rotation as used e.g. in HIGHT. The cipher design aims at small footprint in hardware implementation and small ROM/RAM consumption in software implementation [43]. The best attack on TWINE-80 is a multi-dimensional zero-correlation linear attack on 23 out of 35 rounds [46].

5 Implementation Aspects

AES Size-optimized implementations of AES put the S-box and the round constants in lookup tables as they occupy slightly more than 256 bytes. The code of the size-optimized implementation mostly follows the cipher pseudocode for all three architectures. The speed-optimized implementation for ARM uses the 32-bit lookup tables that combine the S-boxes with the MixColumns transformation (similarly to well-known x86-optimizations). As these tables are too large (15 KBytes for either encryption or decryption) for AVR and MSP, we use only the Galois multiplication table for these architectures. The C code is based on the CryptoLib implementation [39] and the SUPERCOP implementation by Hongjun Wu [9]. An assembly implementation for each target device was considered to explore the performance differences between C and assembly.

Fantomas is implemented to combine lookup-table based linear diffusion layers (so-called L-boxes) with bit-sliced S-boxes, which are computed using a Feistel structure. Storing the 4×512 KB L-boxes in RAM improves the execution time with one quarter on AVR and ARM. Our implementations are based on non-bitsliced C code provided by the designers.

HIGHT is implemented to follow the specifications from [29], which modifies the design from the original paper [30]. The 128 7-bit δ constants are either computed when the key schedule is called or precomputed and stored in Flash or RAM. The fully unrolled version with inlined auxiliary round functions $\mathtt{F}_0$ and $\mathtt{F}_1$ requires half cycles compared to the reference implementation. The cipher byte level rotations on MSP waste half of the microprocessor registers, while on AVR they can be done without penalty. While AVR has 32 general purpose registers, MSP has only 12. For these reasons, the implementation for MSP requires more than three times more clock cycles than the one for AVR.

LBlock is implemented according to the original specifications [48]. 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.

LED is an SPN aimed at very compact hardware implementation. It represents the state by a 4×4 nibble matrix and uses very similar round transformations as the AES, except that it is nibble-oriented. There is no key schedule in LED; the key is simply XORed every 4 rounds. Our implementation of LED combines the `SubSell`, `ShiftRow`, and `MixColumn` operation into a table look-up to reduce execution time.

Piccolo implementation follows closely the cipher description [42]. The arithmetic in GF(2^4) uses only XORs and 2 small look-up tables for multiplications by 2 and 3. The S-box and the key schedule constants are stored in look-up tables. No specific loop unrolling is applied.

PRESENT has a small S-box so its lookup table is used in all implementations. However, its combination with a bit permutation over a 64-bit word is difficult to optimize without using very large (up to 1 MB for decryption) lookup tables. Such tables are affordable only in the speed-optimized implementation for ARM (as in the implementation provided by the BLOC project [16]), whereas for AVR and MSP we had to implement the bit permutation also as a look-up table. The size-optimized implementation follows the cipher's pseudocode and is also taken from [16]. Overall, the bit-oriented structure of PRESENT makes all C software implementations very slow unless they can afford very large lookup tables. We added an assembly implementation which takes advantage of the cipher bit-oriented structure for each target device. The assembly implementation for AVR is around 12 times faster than the C implementation, while the MSP assembly implementation is 19 times faster than the C implementation.

PRINCE is implemented after the original paper [13] and [15]. The optimization strategies considered include using 8, 16, 32 and 64 bit operations were possible and different levels of loop unrolling. The best execution times are obtained using fully unrolled implementation with 8-bit operations for AVR and 16-bit operations for MSP. For ARM the best execution times are achieved using a partially unrolled version with 32-bit oriented operations.

RC5 is implemented by adapting the reference implementation provided in [41]. Because of the simple and efficient design, there are not too many optimization directions. To explore different trade-offs, we fully unrolled the cipher operations and we precomputed the encryption key schedule array `S` and stored it in Flash or RAM.

Robin is implemented in different ways that are based on non-bitsliced C code provided by the 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.

Simon is implemented to optimize for both RAM usage and speed because of Simon's inherent simplicity. It follows the pseudo-code from the specification paper [6]. The rounds are processed by pairs and the rotation functions as well as the Feistel function are inlined. The `Z` constant used in the key schedule is the only lookup table. The C code is written with only 32-bit operations.

Speck is implemented in a straightforward fashion using the pseudo-code from the specifications [6]. It is optimized for both speed and RAM usage. The rounds are processed by pairs and the rotation functions are inlined. The C code uses only 32-bit operations and no look-up table.

TWINE is a very simple cipher so that even the speed-optimized implementation is marginally larger than the size-optimized one. It uses 4-bit branches which, in the authors' implementation [43], reside in separate bytes (so that the entire state is twice as large). We wrote a size-optimized implementation. Both implementations are small enough to run on all platforms. We conclude that TWINE is software-friendly and is one of the easiest ciphers to implement on all platforms.

6 Results

6.1 Methodology

We have gathered and prepared up to 24 implementations for each cipher (more than 80 in total) and benchmarked all of them on each device in each scenario. It is possible to sort all the implementations according to their speed, code, or RAM size in any particular scenario on any device and we maintain a separate interactive web-page [18] where all these orderings can be chosen. Due to space limits for this paper, we have aggregated the data by the following principles, which seem to be the most interesting ones:

– In Scenario 1 we made the full encryption and decryption including the key schedule. Then for each implementation i, and device d we calculate the performance parameter $p_{i,d}$. The value $p_{i,d}$ aggregates the three metrics $M = \{$ the code size, the RAM size, the cycle count $\}$ as follows:

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

where $v_{i,d,m}$ is the value of the metric m for the implementation i on the device d; w_m is the *relative weight* of metric m and $\min_i(v_{i,d,m})$ represents the minimum value of the metric m from all considered implementations of all considered ciphers on the same device d. For each cipher and each device we set $w_m = 1$ (the framework also allows to choose other weights for the metrics; for example the results in Appendix C are computed using different weights w_m for the metrics) and select the implementation with smallest $p_{i,d}$. Finally, for each cipher and the selected set of implementations i_1, i_2, i_3 (one for each device) we calculate the Figure-of-Merit (FOM) value as the average performance value over three devices.

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

Then we sort the ciphers accordingly to FOM (Table 2-I). We also list the encryption and decryption benchmarks alone for the same implementations (Table 2-II,III).

– In Scenario 2, we also select for each cipher and device the best implementation. First, we select the most balanced implementation using Equation (1) and $w_m = 1$ (Table 3). In Table 4-I we calculate $p_{i,d}$ a bit differently:

$$p_{i,d} = \sum_{m \in \{\text{code, RAM}\}} w_m \frac{v_{i,d,m}}{\max_i(v_{i,d,m})}, \tag{3}$$

where $\max_i(v_{i,d,m})$ is the maximum value of Flash (for the code size metric) or RAM (for RAM metric) available on device d (see Section A). Thus we essentially measure the fraction of available memory occupied by the implementation. Finally, in Table 4-II the best implementation for a cipher is the one with the smallest cycle count.

Defining a fair Figure of Merit that considers various trade-offs is a challenging task. The Figure of Adversarial Merit (FOAM) introduced in [33] combines inherent security provided by cryptographic structures and components with their implementation properties allowing the comparison of security-time-area trade-offs of hardware implementations. Although it considers security, it is suitable only for hardware implementations of SPN-based primitives.

6.2 Comparison with other Benchmarks

Many ciphers in our list have been already benchmarked on AVR, MSP, or ARM architectures separately or within some framework. It is difficult to compare the performance numbers between the frameworks and separate implementations because the methodology is different, the optimization efforts vary, and some assembly implementations can be much faster than some of our assembly or C implementations. We believe that the unified methodology allows for a good overview and insight into the relative performance of the lightweight ciphers. As our framework is open, we expect to receive other optimized implementations in the future so that eventually we get closer to the absolute performance values for each cipher.

The most notable differences of our benchmarks with existing implementations on AVR/MSP/ARM are the following:

Table 2. Results for scenario 1 (encryption of 128 bytes of data using CBC mode). For each cipher, an optimal implementation on each architecture is selected.

Cipher	**AVR**			**MSP**			**ARM**			
	Code Size	RAM	Execution Time	Code Code	RAM	Execution Time	Code Size	RAM	Execution Time	FOM
	[Bytes]	[Bytes]	[cycles]	[Bytes]	[Bytes]	[cycles]	[Bytes]	[Bytes]	[cycles]	
I: Encryption + Decryption (including key schedule)										
Speck	**1644**	305	59612	**1342**	300	93239	**792**	356	**19529**	3.5
Simon	2304	380	82085	9104	380	176700	896	428	24019	6.6
AES	4356*	434*	**59085***	3444*	412*	84070*	3928*	500*	70905*	7.2
Robin	4944	271	146149	3170	238	76878	3684	**320**	92132	7.3
Fantomas	5892	**267**	111677	4164	**234**	**57430**	4620	324	70197	7.4
RC5	4574	378	252147	1952	378	482894	1144	432	32903	8.4
LBlock	3104	336	207590	2024	328	313349	2208	598	140595	9.1
HIGHT	2624	347	166480	2370	340	363829	2196	416	173762	9.6
PRESENT	2840*	458*	245853*	2230*	454*	201885*	2528*	526*	270603*	11.3
Piccolo	2672	324	407890	1824	318	349423	1604	430	291401	12.5
PRINCE	5358	374	243396	4174	240	405552	4304	548	202445	12.7
TWINE	4236	646	297265	3796	564	393320	2464	442	257039	13.5
LED	5156	574	2221555	7004	252	2505640	3640	678	585216	44.2
II: Encryption (without key schedule)										
Speck	**628**	**29**	28401	**474**	36	47991	**300**	**48**	**6948**	
Simon	752	40	39313	524	50	86999	328	56	10471	
AES	1708*	54*	**24695***	1312*	44*	33484*	1544*	108*	30251*	
Robin	2920	61	69199	1976	44	39350	2236	128	45926	
Fantomas	2784	53	51471	1954	42	**29038**	2232	128	35724	
RC5	1614	33	75871	492	**32**	174253	352	56	13108	
LBlock	1336	34	102865	808	36	151463	992	236	69360	
HIGHT	1032	36	84081	806	38	183687	764	80	81660	
PRESENT	1240*	31*	121377*	948*	36*	97367*	1128*	68*	130198*	
Piccolo	1250	46	202033	818	48	171143	728	112	140459	
PRINCE	4210	174	121137	3354	48	202279	3588	308	100770	
TWINE	1872	140	148497	1594	114	191063	936	92	126732	
LED	2600	242	1074961	4362	82	1186231	2032	320	279650	
III: Decryption (without key schedule)										
Speck	**770**	**53**	**29346**	**592**	**48**	41618	**432**	**104**	**11864**	
Simon	898	64	40274	660	64	83906	456	112	12457	
AES	2096*	98*	32862*	1902*	76*	49404*	2224*	164*	39765*	
Robin	3048	111	76950	2218	78	37528	2472	160	46206	
Fantomas	3620	107	60206	2722	74	**28392**	2900	164	34473	
RC5	1774	58	76525	646	50	177244	484	112	14218	
LBlock	1442	62	99442	960	54	155426	1124	292	69180	
HIGHT	1146	59	79810	938	52	178418	844	128	90605	
PRESENT	1388*	56*	121906*	1108*	52*	100786*	1304*	124*	138947*	
Piccolo	1444	70	204274	1008	64	176946	940	176	149822	
PRINCE	4352	198	122082	3496	64	203138	3716	372	101638	
TWINE	1900	160	144225	1736	130	195186	1072	152	125841	
LED	3068	280	1146226	3022	86	1318658	2232	384	305106	

* Results for assembly implementations.

- The BLOC [16] project's MSP implementations of HIGHT, LBlock, Piccolo, and Twine are slightly worse than ours, whereas the implementations of AES and PRESENT are much slower.
- The AVR assembly implementations of PRESENT and AES from ECRYPT project [23] and [24] are slightly slower than our assembly implementations, while our C implementation of HIGHT is on par with the assembly implementation of [24] and five times faster than the assembly implementation of [23].

Table 3. Results for scenario 2 (encryption of 128 bits of data using CTR mode). For each cipher, an optimal implementation on each architecture is selected.

Cipher	**AVR**			**MSP**			**ARM**			
	Code Size [Bytes]	RAM [Bytes]	Execution Time [cycles]	Code Size [Bytes]	RAM [Bytes]	Execution Time [cycles]	Code Size [Bytes]	RAM [Bytes]	Execution Time [cycles]	FOM
Balanced (globally efficient)										
Speck	**666**	**54**	3251	**618**	58	6054	**560**	**120**	**925**	3.5
Simon	772	62	5343	732	72	10930	648	128	1406	4.7
AES	1410*	79*	**3175***	1438*	80*	4190*	1628*	196*	3763*	6.2
RC5	1712	58	8449	700	**54**	20543	676	128	1751	6.6
Fantomas	2496	108	5919	1920	78	**3646**	2156	216	4564	8.1
Robin	2530	108	7813	1942	80	4935	2160	216	6195	9.1
LBlock	1440	64	11183	976	58	18988	1268	308	9035	10.3
HIGHT	1202	59	11335	982	60	23016	1056	152	11623	11.0
PRESENT	1416*	**54***	15239*	1244*	58*	12226*	1532*	140*	16919*	12.5
Piccolo	1298	70	25745	966	70	21448	988	184	18418	15.0
TWINE	1528	64	21701	1922	136	23938	1228	180	15703	15.1
PRINCE	4420	68	17271	3418	70	25340	3768	380	12727	17.6
LED	2602	91	143317	4422	104	148334	2212	392	35195	52.4

* Results for assembly implementations.

- The assembly implementation of TWINE [43] is faster than our C implementation up to the factor of four.
- The designers' assembly AVR implementations of Simon and Speck are about two times as small and five times as fast as ours [7].

6.3 Discussion of Results

In Scenario 1 ("bulk encryption"), the Top-7 ciphers based on the FOM score are Speck, Robin, Fantomas, Simon, RC5, LBlock and Hight; all other evaluated algorithms have a FOM score that is more than three times worse than that of Speck. We remind that the FOM score takes into account all three metrics (i.e. execution time, RAM footprint and code size) and does so across three platforms (AVR, MSP, and ARM). Of course, when looking at performance, RAM footprint, or code size individually, or when looking at AVR, MSP, or ARM individually, the specific ranking can differ significantly from the overall ranking based on the FOM score. Furthermore, it has to be taken into account that several (up to 20) different implementations exist for each cipher. Since these implementations are based on different optimization strategies, they can (and usually do) perform differently on the three platforms. Therefore, it happens that one and the same cipher has a worse execution time on 16-bit MSP than on 8-bit AVR (e.g. LBlock, HIGHT), which is not a mistake but simply the result of considering RAM equally important as execution time. On each platform, we collected our benchmarking results using the implementation that achieved the highest FOM score.

When having a closer look at the results on AVR, it turns out that the top-ranked algorithms are very similar in terms of RAM footprint, which means the overall rank is primarily determined by execution time and code size. A somewhat surprising result is that AES is the fastest of all ciphers on AVR, even though it did not make it into the Top-7 because its high performance comes at the expense of very large code size. Robin and Fantomas earned their Top-7 position mainly because of their good execution time, which is only slightly worse than that of AES. The other five ciphers in the Top-7 have the advantage of small code size, but are significantly slower than AES. The situation is somewhat similar on MSP in the sense that the Top-7 ciphers are very close in terms of RAM footprint and AES is again the fastest. Fantomas, Robin, and Speck are the ciphers that come closest to AES in terms of execution time, whereby Robin is not only fast but also features relatively small code size. The execution time of all other ciphers is at least four times worse than that of AES. Finally, on ARM, the winners in the performance competition are Speck, Simon and RC5, which all outperform AES. Interestingly, these three ciphers also have the top positions in terms of code size. This is due to the ARX design strategy with an extremely simple function. All other candidates are much slower, much larger, or both.

Table 4. Results for scenario 2 (encryption of 128 bits of data using CTR mode). For each cipher, an optimal implementation on each architecture is selected.

Cipher	**AVR**			**MSP**			**ARM**		
	Code Size [Bytes]	RAM [Bytes]	Execution Time [cycles]	Code Size [Bytes]	RAM [Bytes]	Execution Time [cycles]	Code Size [Bytes]	RAM [Bytes]	Execution Time [cycles]
I: Small code size & RAM									
AES	1410*	79*	**3175***	1438*	80*	4190*	1628*	196*	3763*
Fantomas	2496	108	5919	1920	78	**3646**	2156	216	4564
HIGHT	1202	59	11335	982	60	23016	1056	152	11623
LBlock	1388	**51**	16537	976	58	18988	1240	172	10236
LED	2602	91	143317	4042	96	694812	2260	216	41758
PRESENT	1416*	54*	15239*	1244*	58*	12226*	920	168	175092
PRINCE	4420	68	17271	3418	70	25340	4124	248	14365
Piccolo	1298	70	25745	966	70	21448	988	184	18418
RC5	924	64	22459	700	**54**	20543	676	128	1751
Robin	2530	108	7813	1942	80	4935	2160	216	6195
Simon	772	62	5343	732	72	10930	636	128	1930
Speck	**666**	54	3251	**618**	58	6054	**560**	**120**	**925**
TWINE	1528	64	21701	1570	72	34778	1228	164	20531
II: Best execution time									
AES	1410*	79*	**3175***	8844	348	**2862**	7332	216	2420
Fantomas	2496	108	5919	1920	78	3646	2060	1208	3535
HIGHT	5738	58	6365	15090	64	19796	6968	152	5878
LBlock	8456	55	10327	10556	64	15212	7712	304	7374
LED	2548	267	135061	4422	104	148334	2212	392	35195
PRESENT	1416*	**54***	15239*	1244*	58*	12226*	3568	2332	16786
PRINCE	11382	94	13469	15728	76	21124	13392	352	11796
Piccolo	1298	70	25745	966	70	21448	988	184	18418
RC5	1712	58	8449	2978	**50**	14410	1592	124	1425
Robin	2530	108	7813	1942	80	4935	2128	1212	5202
Simon	772	62	5343	878	84	9648	648	128	1406
Speck	**666**	**54**	3251	**618**	58	6054	**560**	**120**	**925**
TWINE	1528	64	21701	1922	168	23938	1228	180	15703

* Results for assembly implementations.

The overall ranking in Scenario 2 ("challenge-response authentication") is similar to that of Scenario 1. The top-7 are held by the same ciphers, even though their individual ranking can be different. All algorithms outside the Top-7 are at least four times worse with respect to FOM score than the best algorithm, which is again Speck. RC5 climbed from rank 5 to rank 3, mainly because the round keys are pre-computed in Scenario 2, i.e. no key schedule has to be performed. On the other hand, Robin dropped from 2 to 5, mainly because its RAM footprint is the highest on all three platforms, roughly double than that of Speck. LBlock and HIGHT hold again the last two positions among the Top-7 (similar to Scenario 1) since they are neither particularly fast nor particularly small. The upper part of Table 4 summarizes the results of the implementations with minimal RAM footprint and code size for each of the 13 ciphers. Speck is the most lightweight candidate and, therefore, the best choice for applications where size is the primary constraint. On all three platforms, Speck has a code size of around 600 Bytes and a RAM footprint of less than 100 Bytes. On the other hand, as shown in the lower part of Table 4, when performance is of primary concern and size does not matter much, then AES is a good choice. The execution time of AES significantly improves when using look-up tables to perform the round transformation, which, of course, comes at the expense of increased code size. Also Speck and Fantomas are performance-wise consistently good on all three platforms.

Caveats. The results of any "survey-and-benchmark" paper in lightweight cryptography, including ours, always reflect the state of research at a certain time, namely 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 13 candidates considered in this paper. The AES serves as a good example on how progress in research can yield significantly more efficient implementations. Similar progress could also make one of our lightweight ciphers much faster than anticipated today. This is the very reason why we maintain a web page [18] where readers can find up-to-date benchmarking results and cipher rankings. Furthermore, since we focus on C implementations in this paper, our results reflect, to a certain degree, the state of compiler technology (i.e. the “quality” of GCC for AVR, MSP430 and ARM) at the time when the benchmarks were collected. Also compilers tend to get “better” over time by incorporating more and more advanced code optimization techniques, and this progress may have a certain impact on the execution time of the different ciphers.

Another issue that deservers further elaboration is the “C versus Assembly” question. Even though our toolsuite is also able to benchmark Assembly implementations, we only consider (with two exceptions to be noted below) ciphers written in ANSI C in this paper. We have explained our rationale in favour of C implementations in Section 1. However, there are two exceptions, namely AES and PRESENT, for which we also developed optimized Assembly implementations. The Assembly implementation of AES serves as a good reference since AES is the most widely-used block cipher. On the other hand, we decided to write Assembly code for PRESENT because we were interested in the question of whether a highly-optimized software implementation of PRESENT can compete with more “software-friendly” ciphers. As stated in Section 1, the C language is not well suited for ciphers that perform e.g. multi-word operations or certain bit manipulations. Assembly implementations do not suffer from these limitations and can, therefore, achieve significantly better execution times, especially if these operations are performance-critical. The performance gap between C and Assembly is particularly pronounced on processors that provide specific instructions for such performance-critical operations. For example, the 8-bit AVR platform supports many advanced bit-manipulation instructions that can be used to speed-up Assembly implementations of PRESENT and some other “hardware-oriented” ciphers. Indeed, the Assembly implementation of PRESENT on AVR outperforms its C counterpart by a factor of ten, but is still significantly slower than the top-ranked ciphers.

7 Conclusions

In this paper, we presented a survey and benchmark of 13 lightweight block ciphers based on two usage scenarios that are common for secure communication in the IoT. In particular, we studied their implementation aspects on representative 8-, 16- and 32-bit platforms.

We designed and implemented a benchmarking framework that ensures a fair and consistent evaluation of lightweight block ciphers' performances using the same conditions on the same devices. The metrics (binary code size, RAM footprint and execution time) are extracted using cycle accurate simulators or development boards. For full transparency, the source code of the framework, together with the implementations of the evaluated ciphers are available for free. We strongly encourage the community to use and contribute to our framework, since it allows easy integration and evaluation of new C and assembly implementations. We are committed to maintaining a web page [18] with results obtained by each submitted implementation.

Based on the benchmarking results, we inferred interesting information regarding the link between the design decisions and performance figures. In particular, we confirm that the NSA designs Simon and Speck are among the smallest and fastest ciphers on all platforms. The LS-designs seem to be a promising research direction, but a closer analysis of the security of these constructions is necessary.

Further research may include the addition of new ciphers, integration of countermeasures against physical attacks, extending the framework capabilities to other lightweight symmetric primitives (stream ciphers, hash functions or authenticated encryption) and the integration of other resource-constrained devices.

References

1. F. Abed, E. List, S. Lucks, and J. Wenzel. Cryptanalysis of the Speck family of block ciphers. Cryptology ePrint Archive, Report 2013/568, 2013.
2. Arduino. Arduino Due. `http://arduino.cc/en/Main/arduinoBoardDue`.
3. Atmel. ARM Cortex-M3 datasheet. `http://www.atmel.com/Images/doc11057.pdf`.
4. Atmel. AVR ATmega128 datasheet. `http://www.atmel.com/images/doc2467.pdf`.
5. L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, Oct. 2010.

6. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013.
7. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK block ciphers on AVR 8-bit microcontrollers. Cryptology ePrint Archive, Report 2014/947, 2014.
8. D. Beer. MSPDebug. `http://mspdebug.sourceforge.net/`.
9. D. J. Bernstein and T. Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to/`, Feb. 2015.
10. A. Biryukov and E. Kushilevitz. Improved cryptanalysis of RC5. In *Advances in Cryptology – EUROCRYPT'98*, LNCS 1403, pp. 85–99. Springer, 1998.
11. C. Blondeau and K. Nyberg. Links between truncated differential and multidimensional linear properties of block ciphers and underlying attack complexities. In *Advances in Cryptology – EUROCRYPT 2014*, LNCS 8441, pp. 165–182. Springer, 2014.
12. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems – CHES 2007*, LNCS 4727, pp. 450–466. Springer, 2007.
13. J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger et al. PRINCE – A low-latency block cipher for pervasive computing applications. In *Advances in Cryptology – ASIACRYPT 2012*, LNCS 7658, pp. 208–225. Springer, 2012.
14. C. Boura, M. Naya-Plasencia, and V. Suder. Scrutinizing and improving impossible differential attacks: Applications to CLEFIA, Camellia, LBlock and Simon. In *Advances in Cryptology – ASIACRYPT 2014*, LNCS 8873, pp. 179–199. Springer, 2014.
15. A. Canteaut, T. Fuhr, H. Gilbert, M. Naya-Plasencia, and J.-R. Reinhard. Multiple differential cryptanalysis of round-reduced PRINCE. In *Fast Software Encryption-FSE 2014*, 2014.
16. M. Cazorla, S. Gourgeon, K. Marquet, and M. Minier. Implementations of lightweight block ciphers on a WSN430 sensor. `http://bloc.project.citi-lab.fr/library.html`, Feb. 2015.
17. M. Cazorla, K. Marquet, and M. Minier. Survey and benchmark of lightweight block ciphers for wireless sensor networks. In *Proceedings of the 10th International Conference on Security and Cryptography (SECRYPT 2013)*, pp. 543–548. SciTePress, 2013.
18. CryptoLUX. Lightweight Cryptography. `http://cryptolux.org/index.php/Lightweight_Cryptography`, 2015.
19. J. Daemen and V. Rijmen. *The design of Rijndael: AES - the Advanced Encryption Standard*. Springer, 2002.
20. P. Derbez and P.-A. Fouque. Exhausting Demirci-Selçuk meet-in-the-middle attacks against reduced-round AES. In *Fast Software Encryption – FSE 2013*, LNCS 8424, pp. 541–560. Springer, 2013.
21. 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 *Advances in Cryptology – ASIACRYPT 2013*, LNCS 8269, pp. 337–356. Springer, 2013.
22. T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni et al. Compact implementation and performance evaluation of block ciphers in ATtiny devices. In *Progress in Cryptology – AFRICACRYPT 2012*, LNCS 7374, pp. 172–187. Springer, 2012.
23. T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni et al. Implementations of low cost block ciphers in Atmel AVR devices. `http://perso.uclouvain.be/fstandae/lightweight_ciphers/`, Feb. 2015.
24. T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6):522–533, 2007.
25. 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`, April 2011.
26. V. D. Gligor. Light-Weight Cryptography – How Light is Light? Keynote presentation at the Information Security Summer School, Florida State University. Available for download at `http://www.sait.fsu.edu/conferences/2005/is3/resources/slides/gligorv-cryptolite.ppt`, May 2005.
27. V. Grosso, G. Leurent, F.-X. Standaert, and K. Varıcı. LS-designs: Bitslice encryption for efficient masked software implementations. In *Fast Software Encryption – FSE 2014*, LNCS 8540, pp. 18–37. Springer, 2015.
28. J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw. The LED block cipher. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, LNCS ??, pp. 326–341. Springer, 2011.
29. B Han, H Lee, H Jeong, and Y Won. The HIGHT encryption algorithm. Internet-Draft draft-kisa-hight-00, IETF, 2011. `https://tools.ietf.org/id/draft-kisa-hight-00.txt`.
30. D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong et al. HIGHT: A new block cipher suitable for low-resource device. In *Cryptographic Hardware and Embedded Systems – CHES 2006*, LNCS 4249, pp. 46–59. Springer, 2006.
31. IEEE Standards Association. IEEE 802.15: Wireless Personal Area Networks (PANs). `http://standards.ieee.org/about/get/802/802.15.html`.
32. Texas Instruments. MSP430F1611 datasheet. `http://www.ti.com/lit/ds/symlink/msp430f1611.pdf`.

33. K. Khoo, T. Peyrin, A. Y. Poschmann, and H. Yap. Foam: Searching for hardware-optimal SPN structures and components with a fair comparison. In *Cryptographic Hardware and Embedded Systems – CHES 2014*, LNCS 8731, pp. 433–450. Springer, 2014.
34. G. Leander, B. Minaud, and S. Rønjom. A generic approach to invariant subspace attacks: Cryptanalysis of Robin, iSCREAM and Zorro. In *Advances in Cryptology – EUROCRYPT 2015*, LNCS 9056, pp. 254–283, Springer, 2015.
35. F. Mendel, V. Rijmen, D. Toz, and K. Varc. Differential analysis of the LED block cipher. In *Advances in Cryptology – ASIACRYPT 2012*, LNCS 7658, pp. 190–207. Springer, 2012.
36. National Institute of Standards and Technology (NIST). Lightweight Cryptography Workshop 2015. `http://www.nist.gov/itl/csd/ct/lwc_workshop2015.cfm`.
37. National Institute of Standards and Technology (NIST). SHA-3 Competition. `http://csrc.nist.gov/groups/ST/hash/sha-3/`.
38. N. Wang, X. Wang, K. Jia, and J. Zhao Differential attacks on reduced SIMON versions with dynamic key-guessing techniques. Cryptology ePrint Archive, Report 2014/448, 2014.
39. D. Otte. AVR Crypto Lib. `https://www.das-labor.org/wiki/AVR-Crypto-Lib/en`.
40. NIST. FIPS Pub. 197: Advanced encryption standard (AES). *Federal Information Processing Standards Publication 197*, 2001.
41. R. L. Rivest. The RC5 encryption algorithm. In *Fast Software Encryption*, LNCS 1008, pp. 86–96. Springer, 1995.
42. K. Shibutani, T. Isobe, H. Hiwatari, A. Mitsuda, T. Akishita, and T. Shirai. Piccolo: An ultra-lightweight blockcipher. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, LNCS 6917, pp. 342–357. Springer, 2011.
43. T. Suzaki, K. Minematsu, S. Morioka, and E. Kobayashi. TWINE: A lightweight, versatile block cipher. In *Proceedings of the ECRYPT Workshop on Lightweight Cryptography*, pp. 146–169, 2011.
44. B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora - The AVR Simulation and Analysis Framework. `http://compilers.cs.ucla.edu/avrora/`, 2005.
45. 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*, p. 67. IEEE Press, 2005.
46. Y. Wang and W. Wu. Improved multidimensional zero-correlation linear cryptanalysis and applications to LBlock and TWINE. In *Information Security and Privacy – ACISP 2014*, LNCS 8544, pp. 1–16. Springer, 2014.
47. C. Wenzel-Benner and J. Gräf. XBX: eXternal Benchmarking eXtension for the SUPERCOP crypto benchmarking framework. In *Cryptographic Hardware and Embedded Systems – CHES 2010*, LNCS 6225, pp. 294–305. Springer, 2010.
48. W. Wu and L. Zhang. LBlock: A lightweight block cipher. In *Applied Cryptography and Network Security – ACNS 2011*, LNCS 6715, pp. 327–344. Springer, 2011.
49. P. Zhang, B. Sun, and C. Li. Saturation attack on the block cipher HIGHT. In *Cryptology and Network Security – CANS 2008*, LNCS 5888, pp. 76–86. Springer, 2009.
50. ZigBee Alliance. ZigBee Wireless Standard. `http://www.zigbee.org/`.

A Target Devices

8-bit AVR ATmega128 Microcontroller

The ATmega128 [4] microcontroller manufactured by Atmel uses an 8-bit RISC microprocessor which supports 133 instructions. Most of the instructions are encoded into 16 bits and require a single clock cycle to execute. The two stages, single level pipeline allows the execution of instructions in each clock cycle because while one instruction is executed the next one is fetched from the program memory.

The 32 8-bit general purpose registers (`R0` – `R31`) are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The ALU operations are divided into three main categories: arithmetic, logic and bit-functions. Six of the 32 registers can be used as three 16-bit indirect address register pointers (`X`, `Y` and `Z`) for addressing the data space.

The Harvard memory architecture maximizes performance and parallelism. Memory includes 128 KBytes of Flash, 4 KBytes of SRAM and 4 KBytes of EEPROM for data storage. Each program memory address contains a 16-bit or 32-bit instruction. The data memory supports five different addressing modes: direct, indirect, indirect with displacement, indirect with pre-decrement and indirect with post-increment.

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

16-bit MSP430F1611 Microcontroller

The MSP430F1611 [32] microcontroller produced by Texas Instruments uses a 16-bit RISC architecture with an instruction set consisting of 27 core instructions and 24 emulated instructions. Each instruction uses an even number of bytes (two, four or six). There are three core instructions formats: dual-operand, single operand and jump. The number of clock cycles required by an instruction depends on the instruction format and addressing mode used.

The microprocessor supports seven addressing modes: register, indexed, symbolic (PC relative), absolute, indirect register mode, indirect autoincrement, immediate. It has 16 16-bit registers (`R0 - R15`) from which 12 are general purpose registers (`R4 - R15`). The register operations take one cycle.

The Von Neumann memory consists in 48 KBytes of Flash and 10 KBytes of SRAM. The Flash memory is bit, byte and word addressable and programmable.

This microcontroller's typical applications are sensor systems, industrial control and hand-held meters. It supports five power saving operating modes which can be configured by software.

32-bit ARM Cortex-M3 Microcontroller

The Arduino Due [2] microcontroller is based on the Atmel SAM3X8 32-bit ARM Cortex-M3 [3] processor. The Thumb-2 instruction set used by the 32-bit RISC processor ensures high code density and reduced program memory requirements. The high performance processor core uses a three stage pipeline Harvard architecture. It has 13 32-bit general purpose registers (`R0 - R12`), which can be used for data operations.

The microcontroller provides 512 KBytes of Flash organized in two banks of 256 bytes with 1024 pages each and 96 KBytes of SRAM split in two banks of 64 KBytes and 32 KBytes.

ARM Cortex-M3 is the industry-leading 32-bit processor for highly deterministic real time applications. Characterized by ultra-low power consumption, it is suitable for a wide range of low cost platforms: microcontrollers, automotive systems, industrial control systems, wireless networking and sensor nodes.

B Requirements

To unify evaluation conditions, our framework imposes some requirements on each block ciphers' implementations. Firstly, basic operations must be performed by functions having the following signatures.

```
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 should be implemented. The framework will take the use of a common key schedule into account when computing the different metrics. Secondly, all common code sections should be implemented as distinct functions to reduce the code size. In this case, the implementer has to add the common code files names to the implementation info file, which is parsed by the framework when extracting the metrics for the implementation. Thirdly, we give the implementer the possibility to chose where to store the constants used by the cipher: in Flash/ROM or in RAM. This flexibility has a price: the implementer has to define and use a dedicated macro to read the respective constant value. Fourthly, the implemented lightweight cipher's block size in bits has to be equal to or divide 128. While these requirements are formulated to guarantee the same evaluation conditions for an accurate assessment of block ciphers' performances, they limit the possibility to benchmark highly optimised implementations such as bit sliced versions. Yet our aim is not to assess extreme optimizations, which are likely to never be used in practice because of the unreasonable trade-off (i.e. a very fast cipher implementation that uses all available memory has no real world application since there is no space left for other features).

The framework is able to automatically verify the implementation compliance with the formulated requirements and check the implementation's correctness using the provided test vectors. The metrics extraction process is completely automated and remains easy even for users with little experience. We are committed to maintaining a web page [18] with results obtained for each submitted implementation and strongly encourage the community to contribute with implementations to our framework. We think that a common, open and free environment can create a culture of fair comparison of lightweight block ciphers.

C Different Weights for the Metrics

Table 5. Results for scenario 1 (encryption of 128 bytes of data using CBC mode) when using different weights w_m for the three metrics in Equation (1) to compute the performance parameter p_i, d. Namely, the code size and the RAM size have the weights $w_{code} = w_{RAM} = 1$, while the cycle count has the weight $w_{cycle} = 2$. This means that the execution time is twice as important as the code size or RAM consumption. The Figure-of-Merit (FOM) is computed using Equation (2). For each cipher, an optimal implementation on each architecture is selected.

Cipher	**AVR**			**MSP**			**ARM**			
	Code Size	RAM	Execution Time	Code Size	RAM	Execution Time	Code Size	RAM	Execution Time	FOM
	[Bytes]	[Bytes]	[cycles]	[Bytes]	[Bytes]	[cycles]	[Bytes]	[Bytes]	[cycles]	
I: Encryption + Decryption (including key schedule)										
Speck	**1644**	305	59612	**1342**	300	93239	**792**	356	**19529**	4.7
Simon	2304	380	82085	9398	394	162012	896	428	24019	8.5
AES	4356*	434*	**59085***	3444*	412*	84070*	3928*	500*	70905*	9.2
Fantomas	5892	**267**	111677	4164	**234**	**57430**	4620	324	70197	9.6
Robin	4944	271	146149	3170	238	76878	3684	**320**	92132	10.1
RC5	4574	378	252147	1952	378	482894	1144	432	32903	13.3
LBlock	2954	494	183324	2024	328	313349	2208	598	140595	14.6
HIGHT	2624	347	166480	2370	340	363829	2196	416	173762	15.8
PRESENT	2840*	458*	245853*	2230*	454*	201885*	2528*	526*	270603*	18.5
PRINCE	5358	374	243396	4174	240	405552	4304	548	202445	20.0
Piccolo	2672	324	407890	1824	318	349423	1604	430	291401	21.9
TWINE	4236	646	297265	3796	564	393320	2464	442	257039	22.0
LED	5156	574	2221555	7004	252	2505640	3640	678	585216	82.1
II: Encryption (without key schedule)										
Speck	**628**	**29**	28401	**474**	36	47991	**300**	**48**	**6948**	
Simon	752	40	39313	670	62	76743	328	56	10471	
AES	1708*	54*	**24695***	1312*	44*	33484*	1544*	108*	30251*	
Fantomas	2784	53	51471	1954	42	**29038**	2232	128	35724	
Robin	2920	61	69199	1976	44	39350	2236	128	45926	
RC5	1614	33	75871	492	**32**	174253	352	56	13108	
LBlock	1254	161	89361	808	36	151463	992	236	69360	
HIGHT	1032	36	84081	806	38	183687	764	80	81660	
PRESENT	1240*	31*	121377*	948*	36*	97367*	1128*	68*	130198*	
PRINCE	4210	174	121137	3354	48	202279	3588	308	100770	
Piccolo	1250	46	202033	818	48	171143	728	112	140459	
TWINE	1872	140	148497	1594	114	191063	936	92	126732	
LED	2600	242	1074961	4362	82	1186231	2032	320	279650	
III: Decryption (without key schedule)										
Speck	**770**	**53**	**29346**	**592**	**48**	41618	**432**	**104**	**11864**	
Simon	898	64	40274	808	78	79474	456	112	12457	
AES	2096*	98*	32862*	1902*	76*	49404*	2224*	164*	39765*	
Fantomas	3620	107	60206	2722	74	**28392**	2900	164	34473	
Robin	3048	111	76950	2218	78	37528	2472	160	46206	
RC5	1774	58	76525	646	50	177244	484	112	14218	
LBlock	1390	188	88786	960	54	155426	1124	292	69180	
HIGHT	1146	59	79810	938	52	178418	844	128	90605	
PRESENT	1388*	56*	121906*	1108*	52*	100786*	1304*	124*	138947*	
PRINCE	4352	198	122082	3496	64	203138	3716	372	101638	
Piccolo	1444	70	204274	1008	64	176946	940	176	149822	
TWINE	1900	160	144225	1736	130	195186	1072	152	125841	
LED	3068	280	1146226	3022	86	1318658	2232	384	305106	

* Results for assembly implementations.