ECRYPT II

ICT-2007-216676

ECRYPT II

European Network of Excellence in Cryptology II

Network of Excellence

Information and Communication Technologies

D.SYM.12
Final Lightweight Cryptography Status Report

Due date of deliverable: July 2012
Actual submission date: October 2012

Start date of project: 1 August 2008 Duration: 4 years

Lead contractor: Katholieke Universiteit Leuven (KUL)

Revision 1.0

Project co-funded by the European Commission within the 7th Framework Programme		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission services)	
RE	Restricted to a group specified by the consortium (including the Commission services)	
CO	Confidential, only for members of the consortium (including the Commission services)	

Final Lightweight Cryptography Status Report

Editor
François-Xavier Standaert (UCL)

Contributors
Josep Balasch (KUL), Barış Ege (Raadbout Univ. Nijmegen),
Thomas Eisenbarth (Florida Atlantic University), Benoit Gérard (UCL),
Tim Güneysu (RUB), Stefan Heyse (RUB), Sebastiaan Indesteege (KUL),
Stéphanie Kerckhof (UCL), François Koeune (UCL), Tomislav Nad (IAIK),
Thomas Plos (IAIK), Thomas Pöppelmann (RUB), Francesco Reggazoni (UCL),
François-Xavier Standaert (UCL), Gilles Van Assche (STM), Ronny Van Keer (STM),
Loic van Oldeneel tot Oldenzeel (UCL), Ingo von Maurich (RUB).

October 2012
Revision 1.0

The work described in this report has in part been supported by the Commission of the European Communities through the ICT program under contract ICT-2007-216676. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Contents

Chapter 1

Block ciphers

1.1 Introduction

Small embedded devices (including smart cards, RFIDs, sensor nodes) are now deployed in many applications. They are usually characterized by strong cost constraints. Yet, as they may manipulate sensitive data, they also require cryptographic protection. As a result, many lightweight ciphers have been proposed in order to allow strong security guarantees at a lower cost than standard solutions. Quite naturally, the very idea of "low-cost" is highly dependent on the target technology. Some operations that are extremely low cost in hardware (e.g. wire crossings) may turn out to be annoyingly expensive in software. Even within a class of similar devices (e.g. software), the presence or absence of some options (such as hardware multipliers) may cause strong variations in the performance analysis of different algorithms. As a result, it is sometimes difficult to have a good understanding of which algorithms are actually lightweight on which device. Also, the lack of comparative studies prevents a good understanding of the cost vs. performance tradeoff for these algorithms.

In this paper, we consider this issue of performance evaluation for low-cost block ciphers, and investigate their implementation in ATMEL ATtiny devices [4], i.e. small microcontrollers, with limited memory and instruction set. Despite the relatively frequent use of such devices in different applications, little work has been done in benchmarking cryptographic algorithms in this context. Notable exceptions include B. Poettering's open-source codes for the AES Rijndael [2], and an interesting survey of lightweight cryptography implementations [26]. Unfortunately, these works are still limited by the number of ciphers under investigation and the fact that their source code is not always available for evaluation.

As a result, the goal of this work is to extend the benchmarking of 11 lightweight and standard ciphers, and to make their implementation available under an open-source license. In order to make comparisons as meaningful as possible, we tried to adapt the guidelines proposed in [29] for the evaluation of hardware implementations to our software context. Yet, as the project was involving 11 different designers, we also acknowledge that some biases can appear in our conclusions, due to slightly different implementation choices. Overall, we hope that this initiative can be used as a first step in better understanding the performances of block ciphers in a specific but meaningful class of devices. We also hope that it can be used as a germ to further develop cryptographic libraries for embedded platforms and, in the long term, add security against physical (fault, side-channel) attacks as another evaluation criteria.

The rest of the paper is structured as follows. Section 2 contains a brief description of the implemented ciphers. Section 3 describes our evaluation methodology and metrics. Section 4 provides succinct descriptions of the implementation choices made by our 11 designers. Finally, our performance evaluations are in Section 5. The webpage containing all our open-source codes is given here [1].

1.2 List of Investigated Ciphers

AES Rijndael [20] is the new encryption standard selected in 2002 as a replacement of the DES. It supports key sizes of 128, 192 or 256 bits, and block size of 128 bits. The encryption iterates a round function a number of times, depending on the key size. The round is composed of four transformations: SubBytes (that applies a non-linear S-box to the bytes of the states), ShiftRows (a wire crossing), MixColumns (a linear diffusion layer), and finally AddRoundKey (a bitwise XOR of the round key). The round keys are generated from the secret key by means of an expansion routine that re-uses the S-box used in SubBytes. For low-cost application, the typical choice is to support only the key size of 128 bits.

DESL, DESX, and DESXL [35] are lightweight variants of the DES cipher. For the L-variant, all eight DES S-boxes are replaced by a single S-Box with well chosen characteristics to resist known attacks against DES. Additionally the initial permutation (IP) and its inverse (IP^{-1}) are omitted, because they do not provide additional cryptographic strength. The X-variant includes an additional key whitening of the form: $\mathsf{DESX}_{k,k1,k2}(x) = k2 \oplus \mathsf{DES}_k(k1 \oplus x)$. DESXL is the combination of both variants. The main goal of the developer was a low gate count in hardware implementations similar to the original DES proposal.

HIGHT [33] is a hardware-oriented block cipher designed for low-cost and low-power applications. It uses 64-bit blocks and 128-bit keys. HIGHT is a variant of the generalized Feistel network and is only composed of simple operations: XOR, mod 2^8 additions and bitwise rotations. Its key schedule consists of two algorithms: one generating whitening key bytes for initial and final transformations; the other one for generating subkeys for the 32 rounds. Each subkey byte is the result of a mod 2^8 addition between a master key byte and a constant generated using a linear feedback shift register.

IDEA [34] is a patented cipher whose patent expired in May 2011 (in all countries with a 20 year term of patent filing). Its underlying Lai-Massey construction does not involve an S-box or a permutation network such in other Feistel or common SPN ciphers. Instead, it interleaves mathematical operations from three different groups to establish security, such as addition modulo 2^{16}, multiplication modulo $2^{16}+1$ and addition in GF(2^{16}) (XOR). IDEA has a 128-bit key and 64-bit inputs and outputs. A major drawback of its construction is the complex key schedule involved that also requires the extended Euclidean algorithm for the decryption operation. For efficient implementation, this complex key schedule needs to be precomputed and stored in memory.

KASUMI [3] is a block cipher derived from MISTY1 [39]. It is used as a keystream generator in the UMTS, GSM, and GPRS mobile communications systems. It has a 128-bit key and 64-bit inputs and outputs. The core of KASUMI is an eight-round Feistel network. The round functions in the main Feistel network are irreversible Feistel-like network transformations. The key scheduling is done by bitwise rotating the 16-bit subkeys or XORing them with a constant. There are two S-boxes, one 7 bit and the other 9 bit.

KATAN and KTANTAN [15] are two families of hardware-oriented block ciphers. They have 80-bit keys and a block size of either 32, 48 or 64 bits. The cipher structure resembles that of a stream cipher, consisting of shift registers and non-linear feedback functions. An LFSR counter is used to protect against slide attacks. The difference between KATAN and KTANTAN lies in the key schedule. KTANTAN is intended to be used with a single key per device, which can then be burnt into the device. This allows KTANTAN to achieve a smaller footprint in a hardware implementation. In the following, we considered the implementation of KATAN with 64-bit block size.

mCrypton [37] is a block cipher specifically designed for resource-constrained devices such as RFID tags and sensors. It uses a block length of 64 bits and a variable key length of 64, 96 and 128 bits. In this paper, we implemented the variant with a 96-bit key. mCrypton consists of an AES-like round transformation (12 rounds) and a key schedule. The round transformation operates on a 4×4 nibble array and consists of a nibble-wise non-linear substitution, a column-wise bit permutation, a transposition and a key-addition step. The substitution step uses four 4-bit S-boxes. Encryption and decryption has almost the same form. The key scheduling algorithm generates round keys using non-linear S-box transformations, word-wise rotations, bit-wise rotations and a round constant. The same S-boxes are used for the round transformation and key scheduling.

NOEKEON [16] is a block cipher with a key length and a block size of 128 bits. The block cipher consists of a simple round function that bases only on bit-wise Boolean operations and cyclic shifts. The round function is iterated 16 times for both encryption and decryption. Within each round, a working key is XORed with the data. The working key is fixed during all rounds and is either the cipher key itself (direct mode) or the cipher key encrypted with a null string. The self-inverse structure of NOEKEON allows to efficiently combine the implementation of encryption and decryption operation with only little overhead.

PRESENT [13] is a hardware-oriented lightweight block cipher designed to meet tight area and power restrictions. It features a 64-bit block size and 80-bit key size. PRESENT implements a substitution-permutation network in 32 rounds. The permutation layer consists only of bit permutations. Together with the tiny 4-bit S-box, the design enables minimalistic hardware implementations. The key scheduling consists of a single S-box lookup, a counter addition and a rotation.

SEA [49] is a scalable family of encryption algorithms, defined for low-cost embedded devices, with variable bus sizes and block/key lengths. In this paper, we implemented $\mathrm{SEA}_{96,8}$, i.e. a version of the cipher with 96-bit blocks and keys. SEA is a Feistel cipher that exploits rounds with 3-bit S-boxes, a diffusion layer made of bit and word rotations and a mod 2^n key addition. Its key scheduling is based on rounds similar to the encryption ones and is designed such that keys can be derived “on-the-fly” both in encryption and decryption.

TEA [52] is a 64-bit block cipher using 128-bit keys (although equivalent keys effectively reduce the key space to 2^{126}) . TEA stands for Tiny Encryption Algorithm and, as the name says, this algorithm was built with simplicity and ease of implementation in mind. A C implementation of the algorithm corresponds to about 20 lines of code, and involves no S-box. TEA has a 64-round Feistel structure, each round being based on XOR, 32-bit addition and rotation. The key schedule is also very simple, alternating the two halves of the key at each round. TEA is sensitive to related-key attacks using 2^{23} chosen plaintexts and one related-key query, with a time complexity of 2^{32}

1.3 Methodology and Metrics

In order to be able to compare the performances of the different ciphers in terms of speed, memory space and energy, the developers were asked to respect a list of common constraints, hereunder detailed.

1. The code has to be written in assembly, in a single file. It has to be commented and easily readable, for example, giving the functions the name they have in their original specifications.

2. The cipher has to be implemented in a low-cost way, minimizing the code size and the data-memory use.

3. Both encryption and decryption routines have to be implemented.

4. Whenever possible, and in order to minimize the data-memory use, the key schedule has to be computed "on-the-fly". The computation of the key schedule is always included in the algorithm evaluations.

5. The plaintext, ciphertext and the key have to remain at the same place in the data memory at the beginning and the end of the encryption.

6. The target device is an 8-bit microcontroller from the ATMEL AVR device family, more precisely the ATtiny45. It has a reduced set of instructions and, e.g. has no hardware multiplier.

7. The encryption and decryption routines are called by a common interface.

The SEA reference code was sent as an example to all designers, together with the common interface.

The basic metrics considered for evaluation are code size, number of RAM words, cycle count in encryption and decryption and energy consumption. From these basic metrics, a combined metric was extracted (see Section 1.5). For the energy-consumption evaluations, each cipher has been flashed in a ATtiny45 mounted on a power-measurement board. A 22 Ohms shunt resistor was inserted between the Vdd pin and the 5V power supply, in order to measure the current consumed by the controller while encrypting. The common interface generates a trigger at the beginning of each encryption, and a second one at the end of each of them. The power traces were measured between those two triggers by our oscilloscope through a differential probe. The plaintexts and keys were generated randomly for each encryption. One hundred encryption traces were averaged for each energy evaluation. The average energy consumed by an encryption has been deduced afterwards, by integrating the measured current.

1.4 Implementation Details

AES Rijndael. The code was written following the standard specification and operates on a state matrix of 16 bytes. In order to improve performance, the state is stored into 16 registers, while the key is stored in RAM. Also 5 temporary registers are used to implement the MixColumn steps. The S-box and the round constants were implemented as simple look-up tables. The multiplication operation needed in the MixColums is computed with shift and XOR instructions.

DESXL. In order to keep code size small, we wrote a function which can compute all permutations and expansions depending on the calling parameters. This function is also capable of writing six bit outputs for direct usage as S-box input. Because of the bit-oriented structure of the permutations which are slow in software, this function is the performance bottleneck of the implementation. The rest of the code is written straight forward according to the specification. Beside the storage for plain/ciphertext and the keys $k, k1, k2$, additional 16 bytes of RAM for the round key and the state are required. The S-box and all permutation and expansion tables are stored in Flash memory and processed directly from there.

HIGHT. The implementation choices were oriented in order to limit the code size. First, the intermediate states are stored in RAM at each round, and only two bytes of text and one byte of key are loaded at a time. This way, it is possible to re-use the same code fragment four times per round. Next, the byte rotation at the output of the round function is integrated in the memory accesses of the surrounding functions, in order to save temporary storage and gain cycles. Eight bytes of the subkeys are generated once every two rounds, and are stored in RAM. Finally, excepted for the mod 2^8 additions that are replaced by mod 2^8 subtractions and some other minor changes, decryption uses the same functions as encryption.

IDEA. This cipher was implemented including a precomputed key schedule performed by separate functions for encryption and decryption, respectively, prior the actual cipher operation. During cipher execution the precomputed key (104 bytes) is then read byte by byte from the RAM. The plaintext/ciphertext and the internal state are kept completely in registers (using 16 registers) and 9 additional registers are used for temporary computations and counters. IDEA requires a 16-bit modular multiplication as basic operation. However, in the AVR device used in this work, a dedicated hardware multiplier unit is not available. Multiplication was therefore emulated in software resulting in a data-dependant execution time of the cipher operation and an increased cycle count (about a factor of 4) compared to an implementation for a device with a hardware multiplier.

KASUMI. The code was written following the functions described in the cipher specifications. During the execution, the 16-byte key remains stored in the RAM, as well as the 8-byte running state. This allows using only 12 registers and 24 bytes of RAM. Some rearrangement was done to skip unnecessary moves between registers. The 9-bit S-box was implemented in an 8-bit table, with the MSBs concatenated in a secondary 8-bit table. The 7-bit S-box was implemented in an 8-bit table, wasting the MSBs in the memory. The round keys are derived "on-the-fly". Decryption is very similar to encryption, as usual for a Feistel structure.

KATAN-64[1]**.** The main optimization goal was to limit the code size. The entire state of the cipher is kept in registers during operation. In order to avoid excessive register pressure, the inputs and outputs are stored in RAM, and this RAM space is used to backup the register contents during operation. Only three additional registers need to be stored on the stack. As the KATAN key schedule is computed "on-the-fly", the key in RAM is clobbered and needs to be restored externally for subsequent invocations. In order to implement the non-linear

[1] All six variants of the KATAN/KTANTAN family are supported via conditional assembly. As previously mentioned, our performance evaluations focus on the 64-bit version of KTAN.

functions efficiently, addition instructions were used to compute several logical ANDs and XORs in parallel through carefully positioning the input bits and using masking to avoid undesired carry propagation.

mCrypton. The reference code directly follows the cipher specification. The implementation aims for a limited code size. Therefore, we tried to reuse as much code as possible for decryption and encryption. In addition, we used up to 20 registers during the computations to reduce the cycle count. 12 registers are used to compute the intermediate state and the key scheduling, 6 registers for temporary storage, one for the current key scheduling constant and one for the round counter. After each round the modified state and key scheduling state are stored in RAM. The round key is derived from the key scheduling state and is temporarily stored in RAM. The four 4-bit S-boxes are stored in four 8-bit tables, wasting the 4 most significant bits of each entry, but saving cycle counts. The round constants used in the key scheduling algorithm are stored in an 8-bit table.

NOEKEON. The implementation aims to minimize the code size and the number of utilized registers. During execution of the block cipher, input data and cipher key are stored in the RAM (32 bytes are required). In that way, only 4 registers are used for the running state, one register for the round counter, and three registers for temporary computations. The X-register is used for indirect addressing of the data in the RAM. Similar to the implementation of SEA (detailed below), using more registers for the running state will decrease the cycle count, but will also increase the code size because of a less generic programming. For decrypting data, the execution sequence of the computation functions is changed, which leads only to a very small increase in code size.

PRESENT. The implementation is optimized in order to limit the code size with throughput as secondary criteria. State and round key are stored in the registers to minimize accesses to RAM. The S-boxes are stored as two 256-byte tables, one for encryption and one for decryption. This allows for two S-box lookups in parallel. However, code size can easily be reduced if only encryption or decryption is performed. A single 16-byte table for the S-boxes could halve the overall code size, but would significantly impact encryption times. The code for permutation, which is the true performance bottleneck, can be used for both encryption and decryption.

SEA. The reference code was written following directly the cipher specifications. During its execution, plaintexts and keys are stored in RAM (accounting for a total of 24 bytes), limiting the register consumption to 6 registers for the running state, one register for the round counter and three registers of temporary storage. Note that higher register consumption would allow decreasing the cycle count at the cost of a less generic programming. The S-box was implemented using its bitslice representation. Decryption uses exactly the same code as encryption, with “on-the-fly” key derivation in both cases.

TEA. Implementing TEA is almost straightforward due to the simplicity of the algorithm. The implementation was optimized to limit the RAM usage and code size. As far as RAM is concerned, we only use the 24 bytes needed for plaintext and key storage, with the ciphertext overwriting the plaintext in RAM at the end of the process. The only notable issue regarding implementing TEA concerns rotations. TEA was optimized for a 32-bit architecture and the fact that only 1-position shift and rotations are available on the ATtiny, plus the need to propagate carries, made these operations slightly more complex. In particular, 5-position shifts were optimized by replacing them by a 3-position shift in the opposite direction and recovering boundary carries. Nonetheless, TEA proved to be very easy to implement, resulting in a compact code of 648 bytes.

1.5 Performance Evaluation

We considered 6 different metrics: code size (in bytes), RAM use (in bytes), cycle count in encryption and decryption, energy consumption and a combined metric, namely the code size x cycle count product, normalized by the block size. The results for our different implementations are given in Figures 1.2, 1.3, 1.4, 1.5, 1.6, 1.7. We detail a few meaningful observations below.

First, as our primary goal was to consider compact implementations, we compared our code sizes with the ones listed in [26]. As illustrated in Figure 1.1, we reduced the memory footprint for most investigated ciphers, with specially strong improvements for DESXL, HIGHT and SEA.

Next, the code sizes of our new implementations are in Figure 1.2. The frontrunners are HIGHT, NOEKEON, SEA and KATAN (all take less than 500 bytes of ROM). One can notice the relatively poor performances of mCrypton and PRESENT, which can be explained by the more hardware-oriented flavor of these ciphers. As expected, standard ciphers such as the AES and KASUMI are more expensive, but only up to a limited extent (both are implemented in less than 2000 bytes of ROM).

The RAM use in Figure 1.3 first exhibits the large needs of IDEA regarding this metric (232 words) that are essentially due to the need to store a precomputed key schedule for this cipher. Besides, and following our design guidelines, the RAM use essentially reflects the size of the intermediate state that has to be stored during the execution of the algorithms. Note that for the AES, this is in contrast with the “Furious” implementation in [2], that uses 192 bytes of RAM (it also explains our slightly reduced performances for this cipher).

The cycle count in Figure 1.4 clearly illustrates the performance loss that is implied by the use of simple round functions in most lightweight ciphers. This loss is critical for DESXL and KTAN where the large number of round iterations lead to cycle counts beyond 50,000 cycles. It is also large for SEA, NOEKEON and HIGHT. By contrast, these metrics show the excellent efficiency of the AES Rijndael. Cycle count for decryption (Figure 1.5) shows similar results, with noticeable changes. Most visibly, IDEA decryption is much less efficient than its encryption. The AES also shows non-negligible overhead to decrypt. By contrast, a number of ciphers behave identically in encryption and decryption, e.g. SEA where the two routines are almost identical.

As expected, the energy consumption of all the implemented ciphers (Figure 1.6) is strongly correlated with the cycle count, confirming the experimental results in [23]. However, slight code dependencies can be noticed. It is an interesting scope for research to investigate whether different coding styles can further impact the energy consumption and to what extent.

Eventually, the combined metric in Figure 1.7 first shows the excellent size vs. performance tradeoff offered by the AES Rijndael. Among the low-cost ciphers, NOEKEON and TEA exhibit excellent figures as well, probably due to their very simple key scheduling. This comes at the cost of possible security concerns regarding related key attacks. HIGHT seems to provide a good tradeoff between code size and cycle count. A similar comment applies to SEA, where parts of the overhead comes from a complex key scheduling algorithm (key rounds are as complex as the rounds for this cipher). Despite their hardware-oriented nature, PRESENT and mCrypton offer decent performance in 8-bit devices as well. KATAN falls a bit behind, mainly because of its very large cycle count. Only DESXL appears not suitable for such an implementation context.

1.6 Conclusion

This paper reported on an initiative to evaluate the performance of different standard and lightweight block ciphers on a low cost micro-controller. 11 different ciphers have been implemented with compactness as main optimization criteria. Their source code is available on a webpage, under an open-source license. Our results improve most prior work obtained for similar devices. They highlight the different tradeoffs between code size and cycle count that is offered by different algorithms. They also put forward the weaker performances of ciphers that were specifically designed with hardware performances in mind. Scopes for further research include the extension of this work towards more algorithms and the addition of countermeasures against physical attacks.

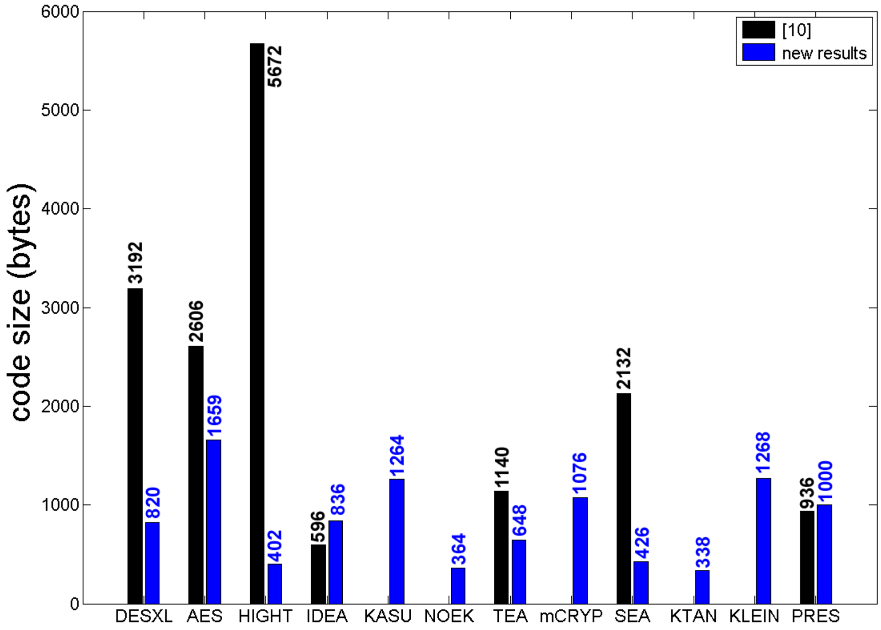

Figure 1.1: Code size: comparison with previous work [26].

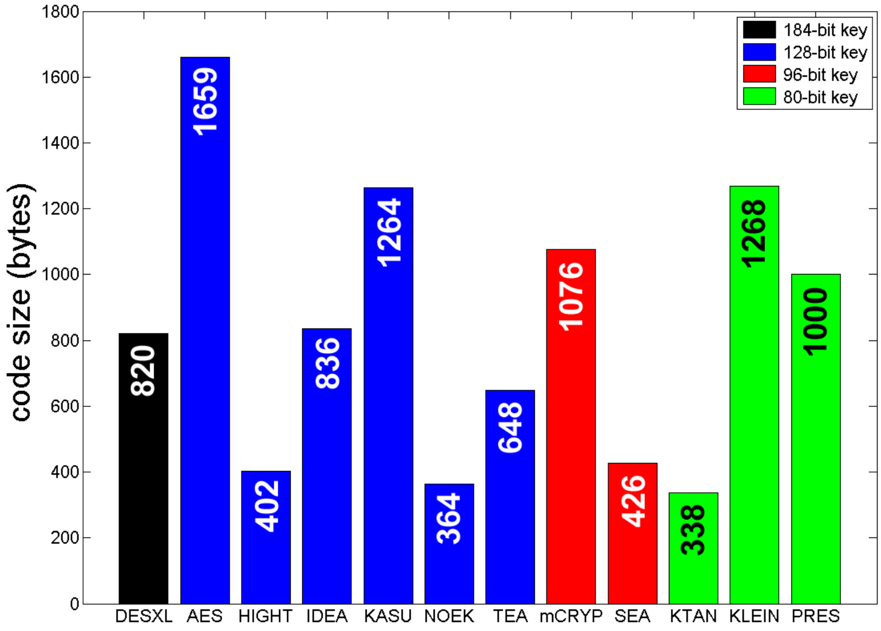

Figure 1.2: Performance evaluation: code size.

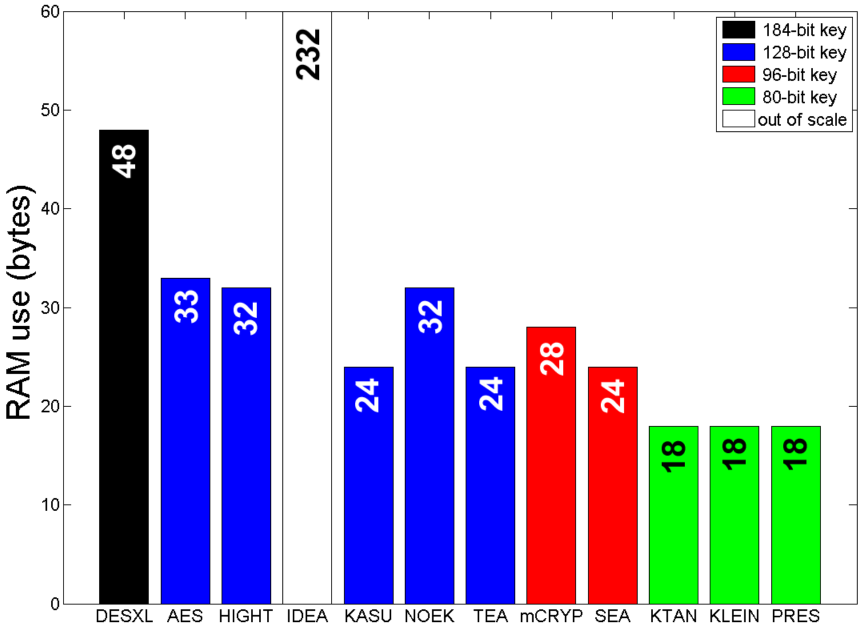

Figure 1.3: Performance evaluation: RAM use.

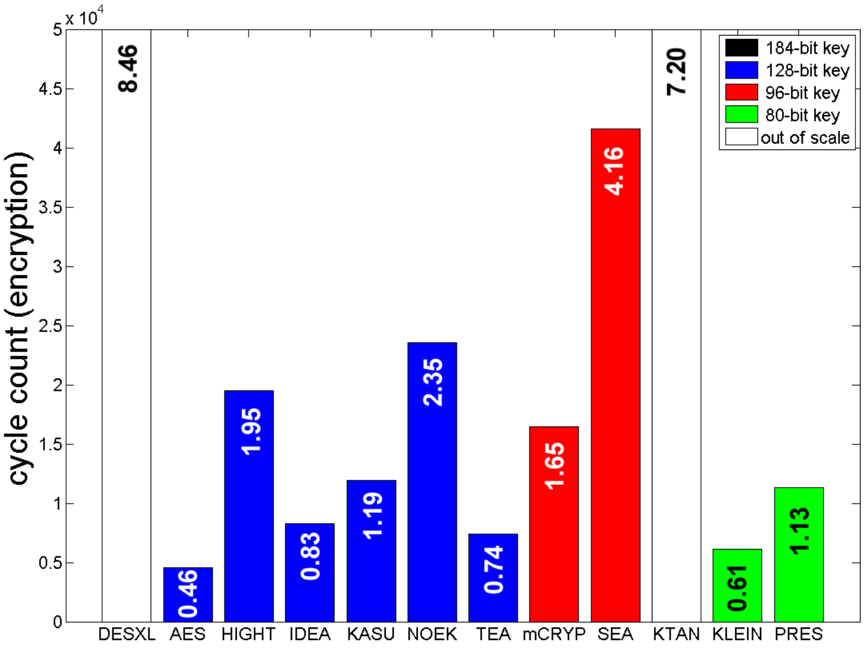

Figure 1.4: Performance evaluation: cycle count (encryption).

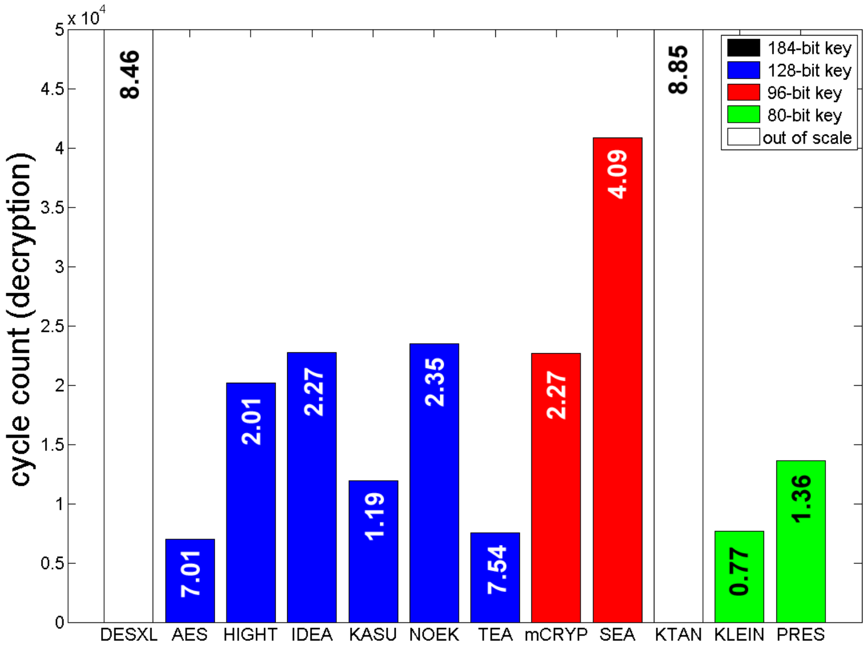

Figure 1.5: Performance evaluation: cycle count (decryption).

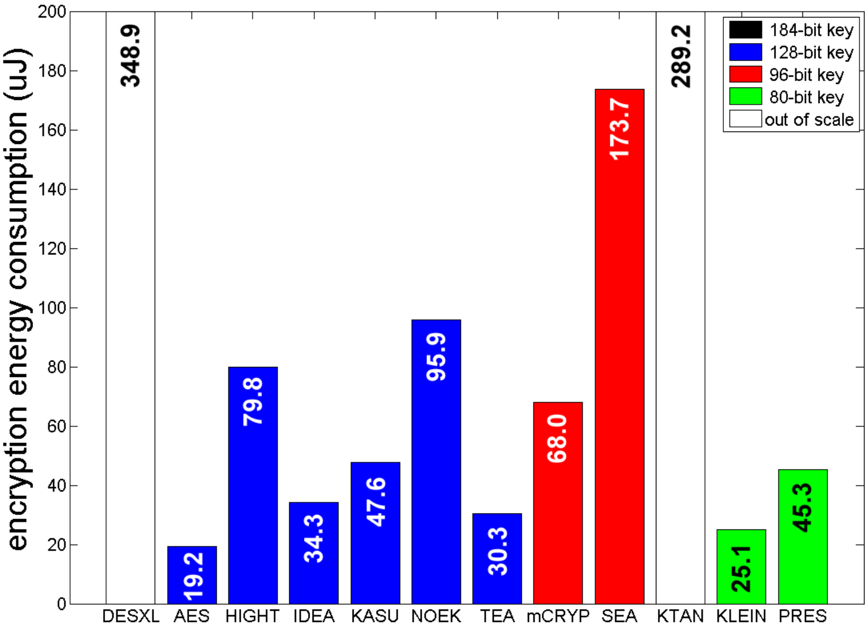

Figure 1.6: Performance evaluation: energy consumption.

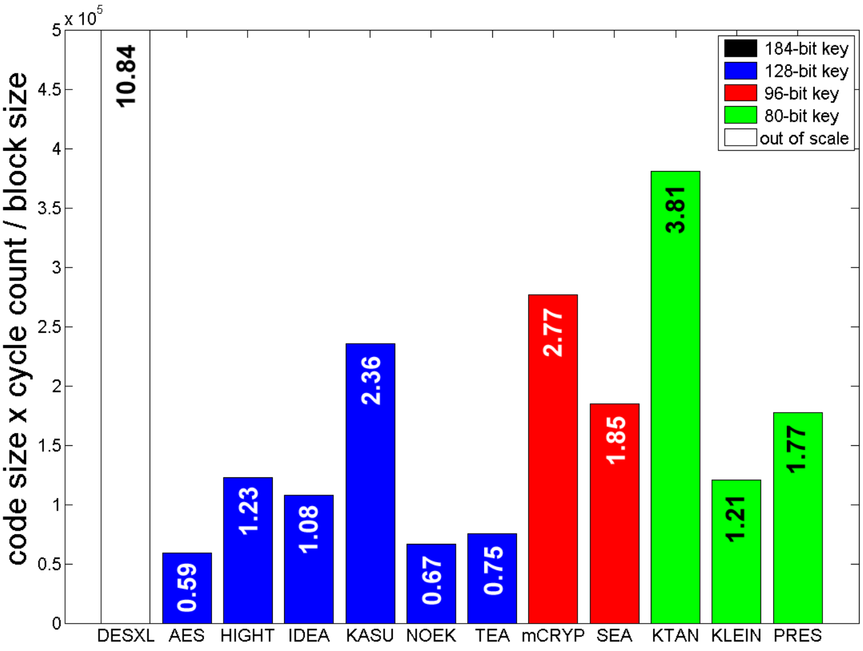

Figure 1.7: Performance evaluation: combined metric.

Chapter 2

Hash functions

2.1 Introduction

Whenever trying to compare different algorithms, such as in the currently running SHA3 competition for choosing a new standard hash function, compact implementations in small embedded devices are an important piece of the puzzle. In particular, they usually reveal a part of the algorithms complexity that does not directly appear in high-end devices, e.g. the need to share resources or to minimize memory. Besides, implementations in small embedded devices such as smart cards, RFIDs and sensor nodes are also motivated by an increasing number of applications. As a result, studying the performances of cryptographic algorithms systematically in this challenging scenario is generally useful.

In a recent work, the implementation of 12 lightweight and standard block ciphers in an ATMEL AVR AtTiny45 has been investigated [24]. In order to increase the relevance of their work, the authors additionally provided open source codes for all their implementations on a public webpage. In this paper, we extend this initiative towards hash functions. For this purpose, we considered three main types of algorithms. First, we targeted SHA256 and the SHA3 finalists. For the latter ones, we only focused on the candidates satisfying the SHA3 security requirements for the 256-bit output length [42], i.e. providing at least 2^{256} (second) preimage resistance and 2^{128} collision resistance. Second, we selected a number of recently published lightweight hash functions, providing both 2^{80} and 2^{128} "flat" security levels[1] [43]. Eventually, we also implemented several block cipher based constructions, e.g. relying on the AES Rijndael.

For all these algorithms, we aimed for the same optimization criteria (namely small source code size and limited memory use) and used a uniform interface (see the details in Section 2.3). Resistance against physical (e.g. side-channel, fault) attacks was explicitly excluded from the requirements. As the project involves many different programmers, we naturally acknowledge possible biases in our performance evaluation results, due to slightly different implementation choices and interpretation of the guidelines. In order to mitigate these (usual) limitations, we also provide all our source codes on a public webpage [1]. As a result, we hope that this initiative can be used as a first step in better understanding the performances of hash functions in a specific but meaningful class of devices.

The rest of the paper is structured as follow. A brief description of all the functions

[1] That is the same security level is required for collision, preimage and second preimage resistance.

implemented is given in Section 2.2. Our methodology and metrics are defined in Section 2.3. The description of the implementation choices for the selected algorithms is in Section 2.5. Eventually, our performance evaluations and the resulting conclusions are in Section 2.6.

2.2 Investigated hash functions

2.2.1 SHA256 and SHA3 candidates

SHA256 is a hash function from the SHA2 family standardized by NIST in FIPS PUB 180-3 [41]. As its predecessor SHA1, it is based on the Merkle-Damgård construction but implements a significant number of changes with respect to SHA1, e.g considering its message expansion, iteration count and an improved compression function. Within the 64 iterations of its compression function, it processes an input block of 512 bits and digests it to an output of 256 bits. Internally, SHA256 also uses a 256-bit state comprising of eight working registers A to H, each of 32 bits in size.

BLAKE-256 operates on 32-bit words and uses the same parameter sizes as SHA256. The input block length is 512 bit and the digest size is 256 bit. A total of $2^{64} - 1$ message bits can be hashed, and BLAKE-256 allows for an optional 128-bit salt. During compression of a message block, a 512-bit internal state is built using a chain value, the salt and the current counter value. Initially, the chain value is set to the same constants as in SHA256. After initialization, 14 message-dependent rounds are applied to the state. Each round consists of eight applications of round function $G_i\,(a, b, c, d)$ which in each iteration transforms 4 out of 16 words of the state using addition modulo 2^{32}, exclusive-or addition, left and right rotations, and memory lookups. In the end, a new chain value is constructed from the internal state, the previous chain value and the salt. Details on BLAKE's inner workings can be found in [7].

Grøstl-256 is an iterated hash function consisting of a compression function built from two distinct permutations, which however share some transformations, called P and Q respectively [30]. These are constructed using the wide-trail design strategy. The hash function is based on a byte-oriented SP-network which borrows components from the AES [19], described by the transforms AddRoundConstant, SubBytes, ShiftBytes and MixBytes. Grøstl is a wide-pipe construction where the size of the internal state (represented by a two 8×16-byte matrices) is significantly larger than the size of the output.

JH-256 is a hash function based on a generalized AES design methodology [54]. It has a 1024-bit state and works on 512-bit input blocks which are processed in three steps. First, the input block is XORed with the left half of the state. A bijective function E8 is then applied to the state. Finally, the right half of the state is XORed with the input block. The bijective function E8 divides the state into an eight-dimensional array on which a substitution-permutation network (SPN) and a maximum-distance seperable (MDS) code are applied.

Keccak is a sponge function family [8, 9]. It uses the sponge construction on top of a permutation $Keccak{-}f[b]$, with the width b chosen between 25 and 1600 by multiplicative steps of 2. Depending on b, the resulting function ranges from a toy cipher to a wide function. The SHA3 candidate Keccak proposes to use exclusively $Keccak{-}f[1600]$ for all output lengths/security levels [10], whereas lightweight alternatives can use for instance $Keccak{-}f[200]$ or $Keccak{-}f[400]$, leaving $Keccak{-}f[800]$ as an intermediate choice [11]. Inside $Keccak{-}f[b]$, the state to process is organized in 5×5 *lanes* of $b/25$ bits each, or alternatively as $b/25$ *slices* of 25 bits each. The round function processes the state using a non-linear layer (χ), a linear diffusion operation (θ), inter- and intra-slice dispersion steps (ρ, π) and the addition of round constants (ι). Details of the design strategy can be found in [9].

Skein-x-y is a cryptographic hash function based on the Threefish block cipher [28]. Its internal state (x) can be 256-, 512- or 1024-bit large, and the output (y) can be of any size. In addition to the simple hashing, it supports a variety of optional features, e.g. MAC and tree-hashing. It is optimized for 64-bit processors. The hash chaining mode of Skein, called UBI, is a variant of the Matyas-Meyer-Oseas hash mode. Threefish is a tweakable block cipher designed for Skein. Its core design principle is based on three operations - XORs, additions and rotations - combined in a large number of rounds (namely 72 or 80). It uses no S-boxes, its nonlinearity comes from alternating additions with XORs. The key schedule generates the subkeys from the key and the 128-bit tweak. We primarily focused on Skein-512-256 which is one of the versions submitted to the NIST SHA3 competition.

2.2.2 Lightweight hash functions

S-Quark and D-Quark are hardware-oriented hash functions with respective digests of length 256 and 176 bits [5]. This family is based on the sponge construction with respective widths 256 and 176 bits and rates 32 and 16 bits. The absorbing phase consists in XORing the message with a part of the state and applying a permutation P. The digest is obtained by squeezing 32/16 bits of the state, then applying the permutation P, and so on until the correct digest size is obtained. The permutation P is composed of 1024/704 iterations of an update function that essentially consists in performing linear retro-action on the state (the polynomials used slightly differ from a version to another). Note that the capacity of S-Quark is only $c = 224$, leading to 112-bit security level (instead of 128 for other lightweight hash functions in our evaluations).

PHOTON is a sponge-based, lightweight, and hardware-oriented hash function family introduced in 2011 [31]. The internal state is represented as a matrix with 4- or 8-bit entries depending on which of the five PHOTON flavors has been selected. Each PHOTON round consists in XORing the message into the state (absorbing) and applying a permutation P twelve times. The first step when applying P is AddConstants where round constants are XORed to the first column of the state. Then, in the SubCells layer, the PRESENT S-box (PHOTON-160/36/36) or AES S-box (PHOTON-256/32/32) is applied to every entry of the state. During ShiftRows the rows of the state matrix are rotated. In the MixColumnsSerial layer a flavor-specific MixColumns matrix is applied to every column of the state several times. This strategy results in a much more compact hardware implementation compared to, e.g. the AES MixColumns layer. When all blocks of the message have been absorbed, a flavor-specific number of squeezing iterations are performed. In each iteration, a small amount of the state is extracted as part of the new hash value and the permutation P is applied twelve times.

SPONGENT is a family of lightweight hash functions based on a wide PRESENT-type permutation [12]. It relies on a sponge construction: a simple iterated design that takes a variable-length input and can produce an output of an arbitrary length n based on a permutation π_b operating on a state of a fixed number b of bits. The different variants are referred to as SPONGENT-$n/c/r$ for different hash sizes n, capacities c, and rates r. Out of the 13 proposed variants, we implemented SPONGENT-160/160/80 and SPONGENT-256/256/128.

Keccak also proposes lightweight alternatives, different from the SHA3 submission, as explained in Section 2.2.1.

2.2.3 Block cipher based constructions

Rogaway-Steinberger LP/lp362 is a hash function construction based on a fixed-key block cipher. The construction principle was developed by Rogaway and Steinberger in 2008 [46] and defines a so-called linearly-determined, permutation based (LP) compression function. An LP^{A}_{mkr} compression function operates on a matrix A (satisfying a special independence criterion) and k permutations, turning an input block of mn bits into an output block of rn bits. The parameter n gives the bit width of the block cipher. We used the block cipher NOEKEON [17] for realizing the fixed-key permutations, for which n equals 128 bits. An LP362 compression function converts a 384-bit input block into a 256-bit output block by using six fixed-key permutations. In case of the LP362 scheme, every permutation uses a different key, whereas in the lp362 scheme, all permutations use the same key (according to [46] both have similar security bounds). XOR operations and finite-field multiplications are used for combining the output of the individual permutations into a single value. The compression function is turned into a hash function using the Merkle-Damgård [22, 40] construction.

Hirose double block length (DBL) construction is a hash function based on a block cipher whose key size exceeds its block size. It has been proposed by Hirose in 2006 [32]. Unlike most previous DBL hash function constructions, Hirose achieves almost optimal collision resistance (if instantiated with an ideal cipher) [32]. For each iteration of the compression function it executes two instances of the block cipher with the same key. The shared key for the parallel encryptions can be used to achieve a performance gain, since only one key schedule is necessary per iteration. The construction requires the key size k to be bigger than the block size n. In fact, the difference between key and block size $n - k$ determines the input size of the compression function, and thus determines the efficiency of the compression function. The output size is $2n$. With AES-256 as the instantiated block cipher (which will be our choice), Hirose DBL has an output size of 256 bit and the compression function takes 128 bit of input per iteration.

The **Davies-Meyer construction** is the most famous cipher-based construction for compression functions. Let a block cipher $E(K, P)$ encrypt the plaintext P using the key K. Then, Davies-Meyer construction updates a chaining value H_i according to a message block M as follows: $H_{i+1} = E(M, H_i) \oplus H_i$. Two instantiations of this construction, using the ciphers Rijndael-256/256 and SEA which are later described, have been studied.

Shrimpton-Stam construction. Based on the proof that compression function constructions relying on PRPs cannot reach optimal security using less than 3 permutations, Shrimpton and Stam proposed a construction relying on 3 different PRPs [48] where the chaining value H_i is updated according to a message block M in the following way $H_{i+1} = f_3\left(f_1(M) \oplus f_2(H_i)\right) \oplus f_1(M)$. They mention that these permutations can be instantiated by a cipher (using three different keys) but only in plaintext feedback mode (that is, the same as in the Davies-Meyer construction). A single instantiation based on Rijndael-256/256 has been considered here.

Rijndael-256/256 is a member of the well-known block cipher family among which the AES standards were chosen [18]. Unlike AES-256, which has a 256-bit key, but a 128-bit state, this 256/256 version processes 256-bit key and state. It is thus a good candidate to instantiate pseudorandom permutations in hash function constructions such as Davies-Meyer or Shrimpton-Stam. Rijndael-256/256 encryption iterates a round function 14 times. This round is composed of four transformations: SubBytes (that applies a non-linear S-box to the bytes of the states), ShiftRows (a wire crossing), MixColumns (a linear diffusion layer), and finally AddRoundKey (a bitwise XOR with the round key). The round keys are generated from the secret key by means of an expansion routine that re-uses the S-box.

SEA is a scalable family of encryption algorithms, designed for low-cost embedded devices, with variable bus sizes and block/key lengths [49]. In this paper, we focus on $\text{SEA}_{192,8}$, i.e. a version of the cipher with 192-bit block and key size. SEA is a Feistel cipher that exploits rounds with 3-bit S-boxes, a diffusion layer made of bit and word rotations and a mod 2^n key addition. Its key scheduling is based on rounds similar to the encryption ones and is designed such that keys can be derived “on-the-fly” both in encryption and decryption.

2.3 Methodology and metrics

In order to be able to compare the performances of the different hash functions in terms of speed and memory space, the developers were asked to respect a list of common constraints, detailed hereunder.

1. The code has to be written in assembly, if possible in a single file. It has to be commented and easily readable, for example, giving the functions the name they have in their original specifications.

2. The function has to be implemented in a low-cost way, minimizing the code size and the RAM use.

3. Data does not have to be preserved by the hashing process. This allows direct modification of the data zones in RAM, hence reducing the amount of memory needed.

4. The interface should be made up of 3 functions. (1) *init* takes no input and initializes the internal state, which is a dedicated memory zone seen as a black box, and returns no output; (2) *update* takes as input a full block of data, updates its internal state by processing that block and returns no output; (3) *final* takes as input the (possibly empty) last chunk of data together with its size and processes it before finalizing the hash computation. By convention, the data passed to *final* is necessarily an incomplete block.

5. Data exchanges are performed with pre-defined memory zones where data has to be put before calling functions, or can be found on their return. For example, the data block to hash has to be put at the pre-defined address *SRAM_DATA* before a call to *update*, and the final hash can be found at *SRAM_STATE* on return of *final*. Most input/output values are thus implicitly passed. The only explicitly passed value is the size of the data passed to *final*.

6. Only the internal state is preserved between calls to these functions. No assumption can be made that other RAM zones (e.g. *SRAM_DATA*) or registers will stay unchanged.

7. The target device is an 8-bit microcontroller from the ATMEL AVR device family, more precisely the ATtiny45. It has a reduced set of instructions and, e.g. has no hardware multiplier.

A common interface file was provided to all designers (available on [1]). Note that for some functions (e.g. for block cipher based), the padding was not explicitly defined. In these cases, we appended n null bytes, followed by the length of the message coded as a 64-bit value, where n is chosen to make the global message length a multiple of the block size.

The basic metrics considered for evaluation are code size, number of RAM words, and cycle count. From these basic metrics, combined metrics were extracted (see Section 2.6). Performances were measured on 4 different message lengths: 8, 50, 100 and 500 bytes, ranging from a very small (smaller than one block) to a large message.

Note finally that, as mentioned in introduction, all hash functions were implemented by different designers, with slightly different interpretations of the low-cost optimizations. As a result, some of the guidelines were not always followed, because of the cipher specifications making them less relevant (which will be specified when necessary).

2.4 Description of the AtTiny45 microcontroller

The ATtiny45 is an 8-bit RISC microcontroller from ATMEL's AVR series. The microcontroller uses a Harvard architecture with separate instruction and data memory. Instructions are stored in a 4 kB Flash memory (2048×16 bits). Data memory involves the 256-byte static RAM, a register file with 32 8-bit general-purpose registers, and special I/O memory for peripherals like timer, analog-to-digital converter or serial interface. Different direct and indirect addressing methods are available to access data in RAM. Especially indirect addressing allows accessing data in RAM with very compact code size. Moreover, the ATtiny45 has integrated a 256-bytes EEPROM for non-volatile data storage.

The instruction-set of the microcontroller contains 120 instructions which are typically 16-bits wide. Instructions can be divided into arithmetic logic unit (ALU) operations (arithmetic, logical, and bit operations) and conditional and unconditional jump and call operations. The instructions are processed within a two-stage pipeline with a pre-fetch and an execute phase. Most instructions are executed within a single clock cycle, leading to a good instructions-per-cycle ratio. Compared to other microcontrollers from ATMEL's AVR series such as the ATmega devices, the ATtiny45 has a reduced instruction set (e.g. no multiply instruction), smaller memories (Flash, RAM, EEPROM), no in-system debug capability, and less peripherals. However, the ATtiny45 has lower power consumption and is cheaper in price.

2.5 Implementation details

2.5.1 SHA256 and SHA3 candidates

SHA256. Like its predecessor SHA1, SHA256 is optimized for 32-bit software implementation. Hence, it can be expected to be similarly efficient on 8-bit AVR processors. Implementing the iteration step of its compression function, a main observation is that six out of eight working registers are just circularly copied. To reduce code and clock cycles for such memory transfer operations, register name reassignment by circular pointer arithmetic is performed instead on the working registers residing in 256 bits of RAM. Circular pointer arithmetic as part of the iteration step is likewise used to update the input word according to the message expansion. Besides 32-bit modular additions, SHA2 requires 32-bit right rotations by $r = \{2, 6, 7, 11, 13, 17, 18, 19, 22, 25\}$ bits and right shifts by $s = \{3, 10\}$. Rotations and shifts by parameters larger than 8 bits first swap 8-bit register accordingly; then single bit operations on the swapped 32-bit word are performed to correspond to $f = \{r, s\} \bmod 8$. SHA256 uses up to three 32-bit bit rotations processing the same input in a row so that reordering of rotation and shift operations by ascending f-values improves efficiency.

BLAKE-256. The RAM consumption is mainly due to storing 64 byte input data, 64 byte state, 32 byte chain value, 8 byte salt, and an 8 byte counter. The initialization vectors (32 byte) and constants (64 byte) are stored in the flash memory of the microcontroller. We refrained from transferring the constant table into the RAM in order to keep RAM consumption low. BLAKE's permutation table σ consists of 10×16 entries. However, each entry is only a four bit number so we merged two entries in one byte and later select the upper/lower 4-bits by masking. Thus, the permutation table requires just 80 byte of ROM instead of 160 byte. In order to maintain a decent performance while keeping the code size down we incorporated the observation made by Osvik [44] which efficiently loads and stores in-/outputs of the round function $G_i\,(a, b, c, d)$. Furthermore, we use loops where applicable and move recurring duties such as loading and storing the counter into functions. An exception to this rule is the implementation of the round function. Since it is called 80 times when hashing one message block its runtime heavily impacts the overall performance. Therefore, we decided to unroll critical parts of the round function.

Grøstl-256. Grøstl has a state of 64 bytes. During the update function, we need to keep the state, the input message and the previously computed hash in memory. Thus, we need 192 byte of RAM. The ShiftBytes is computed by offloading each row, one at a time, from the state into the register of the microcontroller and then writing it back in the new position. In order to increase the performances and reduce the number of accesses to the memory, the SubBytes is computed together with the ShiftBytes. The MixBytes is computed as proposed by Johannes Feichtner [27, 47], and is carried out one column at a time. Finally, to easily compute the padding, 8 bytes of memory are used to keep track of the numbers of messages. This 8 bytes, are copied directly in the appropriate position of the padding block.

JH-256. Specifications for a bitsliced implementation of JH are available, but they suppose that 42 256-bit round constants can be stored in memory, which is not compliant with our low-cost constraints. Hence,JH was implemented according to the reference specifications. The utilisation percentage of the RAM is high as JH needs 128 bytes to store the state, 64 for the input block and 32 for the round constant. In order to improve the performances, the S-box and linear transformation were combined into two look up tables, of 32 bytes each, as was done in the optimized 8-bit implementation provided by JH author [53]. For the same reason, the initial state was precomputed and stored in program memory. It allows us to save the initialisation phase which is equivalent to the processing of one input block. Regarding the permutation, it is performed by reading the states bytes in a different order at the beginning of each round. Finally, state bits are reorganised at the begining and end of each function E8. This bitwise permutation is time consuming and requires additional memory. Those problems can be partially prevented by reorganizing the input bytes before XORing them with the state.

Keccak. In a first level, we implemented the sponge construction, which comes down to XORing r-bit message blocks into the state, with $r > 0$ the *rate*, and to calling the underlying permutation. In a second level, we implemented the permutations $Keccak{-}f[b]$ for $b \in \{200, 400, 800, 1600\}$. The sponge construction imposes that the *capacity* c is twice the security strength level and that $b = r + c$, and our implementation allows any combination of rate and capacity under these constraints. For clarity, the benchmark focuses on three specific instances: the SHA3 candidate *Keccak*$[r = 1088, c = 512]$, and the lightweight variants *Keccak*$[r = 144, c = 256]$ and *Keccak*$[r = 40, c = 160]$ for the 128-bit and 80-bit security strengths levels, respectively. Any pair of instances with $c = 256$ and $c = 160$ would have satisfied the requirements, but our choice aims at minimizing b for a given c and thereby the RAM usage, consistently with a lightweight context. Variants with other RAM usage/code size/cycle count trade-offs can be found in Table 4. Inside the implementation, some operations (i.e., the rotations in θ and ρ) are performed on a lane basis, mapping a lane to $b/200$ byte(s). Some other operations, such as χ or the parity computation in θ, are instead slice-oriented, taking advantage of the representation of 8 consecutive slices in 25 bytes [11]. Note that in the specific case of $Keccak{-}f[200]$, the two approaches collide as the state contains exactly 8 slices or 25 lanes, mapped to 25 bytes. RAM usage is composed of $b/8$ bytes for the state and some working memory ($b/40$ bytes, or 0 for $Keccak{-}f[200]$ as the AVR registers suffice). If the desired output length is greater than the rate (e.g. for lightweight instances), an additional output buffer is needed to perform the squeezing phase.

Skein-x-y. We implemented the SHA3 finalist Skein-512-256, with an output of 256 bits, limited to the hashing functionality. The internal state is therefore made of eight 64-bit words. To keep the program memory space small and the code readable, some basic 64-bits functions like loading, saving, adding, ..., have been coded. The registers are only used temporarily, except the round counter. The message, the state, the key, the key-schedule and the tweak are always in the data space, and modified directly. The three main Threefish functions (addkey, mix and permute) were implemented following the reference specifications. Besides, the modulo 3 and modulo 9 values used in the key schedule were saved in the program memory space. We have also developed Skein-256-256, slightly optimized for the speed and data memory space performances, by leaving most of the time three out of the four state words in the registers.

2.5.2 Lightweight hash functions

S-Quark and D-Quark. The critical point in the implementation of QUARK hash functions is the update of the state[2]. This update phase considers the state as two LFSRs that will be updated using three retro-action polynomials[3]. This design is thought for hardware, a context where it is very efficient, but is much more expensive in software. Nevertheless, our choice to implement this step using a bit-slice approach provides rather good performances. The platform is an 8-bit microprocessor and the retro-action polynomials are such that the last 8 bits of each LFSR are not considered. Hence, our implementation performs 8 updates

[2] During implementation, a minor inconsistency was discovered between the paper description [5] and the reference code [6], which use different bit ordering conventions. We chose to comply with the description provided in the original article. Compliance with the C code can be obtained by inverting the order of bits in the input message.

[3] An additional third will provide constants for the 1024/704 executions required to apply the permutation P.

at the same time reducing from 1024/704 to 128/88 polynomial computations. The state is stored in RAM, as it is too large to be kept in registers. Computations are ordered in such a way that the shift of the state is performed on the fly.

PHOTON-160/36/36 and PHOTON-256/32/32. First note that these implementation significantly differ, since PHOTON-160 has a state matrix with 4-bit cells and uses the PRESENT S-box while PHOTON-256 has 8-bit entries and uses the AES S-Box. This results in different implementation strategies.

The state of the implemented PHOTON-160/36/36 variant consists of 7-by-7 4-bit elements which are packed into 25 bytes in order to save memory. This allows an optimal usage of the RAM but naturally also results in additional code in order to extract the correct nibble out of the state. It is a trade-off between code size/speed and RAM usage. As the interface only allows messages that are a multiple of 8 bits while each iteration of a PHOTON-160/36/36 round function absorbs 36 bits, we just process an input block of length 72 bits and call the PHOTON round function internally twice for a full 72-bit block. The largest amount of computational time is spend in the permutation layer for ShiftRows and especially during the MixColumnsSerial step as finite field arithmetic has to be carried out on 4-bit values.

The internal state of PHOTON-256/32/32 consists of 36 bytes, arranged as a 6-by-6 matrix, that goes over four different transformations to produce a 32 byte hash digest. Due to their sizes both state and digest have to be stored in SRAM. This generates an inherent implementation overhead, as state bytes need to be fetched from and stored to SRAM once for each transformation. We partially reduce this overhead by merging all row-based transformations, and also by incrementing code size. Due to its use of AES-like permutations, the implementation of the PHOTON-256/32/32 transformations can be carried out quite efficiently on 8-bit controllers. The SubCells transformation is implemented as a memory aligned lookup table resulting in important cycle savings. The MixColumnsSerial transformation, consisting of six consecutive calls to the AES MixColumns transformation, is similarly optimized by implementing the multiplication by ‘02’ as a memory aligned lookup table [19].

SPONGENT-160/160/80 and SPONGENT-256/256/128. The SPONGENT-160 state is $160 + 80 = 240$ bits or 30 bytes large. Therefore, the state can be stored in the registers already available on the target device. However, SPONGENT uses a PRESENT-like bit permutation in π_b and therefore every output bit of an S-Box is mapped to a distinct nibble after permutation. If we were to store the state in the available registers, we would only have two registers for additional computations and this would lead to a large code size when implementing the bit permutation. Therefore, the state is stored in SRAM and a three-step iterative approach is used for the bit permutation to achieve a smaller code size. For the permutation, each four consecutive nibbles are permuted and stored in SRAM at the same places. Then, the permuted nibbles are re-ordered to obtain permuted bytes and finally bytes are re-ordered to their appropriate places in the state. Although this approach is code-size efficient, note that it leads to an increase in running time of the overall hashing process. The remaining operations like round constant computation, padding and control logic are implemented in a straightforward manner.

The state of SPONGENT-256 is $256 + 128 = 384$ bits or 48 bytes large. Since the state does not fit into the available registers, we optimized this variant with respect to code size and the state is kept in SRAM. For the permutation, iteratively four successive bytes are loaded into registers and the permuted byte is constructed from two bits at fixed offsets of each of these four bytes. Afterwards the processed bytes are stored back to SRAM. This method keeps the code very small but requires a copy of the 48 bytes state and therefore doubles the required memory. Besides the two states no additional memory is required. The S-Boxes are stored in Flash memory and must be aligned to a address dividable by 16 for easier pointer arithmetic. Again, the remaining operations are straightforwardly implemented.

2.5.3 Block cipher based constructions

Rogaway-Steinberger LP/lp362. For realizing the Rogaway and Steinberger construction principle, the matrix A suggested by Lee and Park [36] with $\alpha = 2$ has been used. For operations in $F_{2^{128}}$ (addition and multiplication) we have selected the same irreducible polynomial $x^{128} + x^7 + x^2 + x + 1$ as stated in [46]. The implementation of the block cipher NOEKEON is based on the open source version published in [24], but the decryption functionality has been removed since it is not required for the generation of a permutations. Two variants of the Rogaway-Steinberger scheme have been implemented: LP362 and lp362. The two variants mainly differ in code size. The lp362 scheme uses a single fixed key for all permutations, leading to about 100 bytes less code than for the LP362 scheme which uses a different fixed key for each of the six permutations. Both variants have similar execution time, consume 92 bytes of RAM, and make use of 8 registers for computing the hash value of a message.

Hirose double block length (DBL) construction For simplicity we chose an all-zero IV and the additive constant to be 1. One of the advantages of Hirose is that the two parallel AES executions use the same key. However, due to memory restrictions, the key should be computed on-the-fly. Hence, the two encryptions need to be processed in parallel. The AES design follows the same design paradigm as the AES presented in [24], with a further optimized `Shift_Rows` operation. Decryption code is not needed and has been removed. The key scheduling is performed on-the-fly and and processes 32 bit at a time. The full 128-bit state of one encryption block is kept in the registers. Since both encryptions are performed in parallel, the two states have to be swapped in and out of SRAM regularly. Due to the large key size, the swap is performed as little as every 4 rounds, keeping the resulting overhead at a minimum. The implementation needs 82 bytes of RAM. We chose not to overwrite the input to the update function, which results in a need for 16 additional RAM bytes for the input. By overwriting the input these additional 16 bytes can be saved if RAM size is critical.

Davies-Meyer construction. The implementation of the Davies-Meyer construction simply requires making a copy of the message to be XORed with the resulting encryption, resulting in an additional consumption of 32 bytes of RAM.

Shrimpton-Stam construction. The implementation of Shrimpton-Stam construction only requires to take care of remembering inputs of the ciphers to be able to XOR them to the result of the encryptions. We chose simple keys to instantiate the functions f_i so that no extra memory is required to store them. More precisely, we respectively set all key bytes to `0x00, 0x11` and `0x22` for f_1, f_2 and f_3.

Rijndael-256/256. The operations to be performed during a Rijndael-256/256 encryption are simple and can be made efficient using the well-known techniques for implementing AES on lightweight processors, like the use of a lookup table for the S-box and the efficient multiplication by '02' for MixColumns [21]. The main issue when working on an ATtiny45 is the state size: whereas AES state can be kept in registers, this is not possible any more for 256-bit blocks. As RAM accesses are time-consuming on the ATtiny, the design of this implementation focuses on minimizing the number of these accesses. This has been done by reorganizing the round loop (without, of course, affecting the behaviour of the cipher) in such a way that the round ends with a ShiftRows operation. Additionally, we used an auxiliary state to perform ShiftRows efficiently. As a result, we can fetch a full column from RAM, immediately perform MixColumns, AddRoundKey and SubBytes, and write the result in the auxiliary RAM state, taking the effect of ShiftRow into account to determine the exact locations in RAM. The next round is then performed similarly, but writing data from the auxiliary state to the initial one, and so on.

SEA. The reference code was written following directly the cipher specifications, and is a natural extension of the 96-bit version designed in [24]. During its execution, plaintexts and keys are stored in RAM (accounting for a total of 48 bytes), limiting the register consumption to 12 registers for the running state, one register for the round counter and some additional temporary storage. The S-box was implemented using its bitslice representation. The block cipher was then inserted in a Davies-Meyer mode of operation, using a similar code as the version using Rijndael-256/256. Overall, the implementation maintains low code size and RAM use at the cost of a large cycle count, mainly due to the large number of rounds (177) in the 196-bit version of the cipher based on 8-bit words.

2.6 Performance evaluation & conclusions

We first refer to a number of other implementations of hash functions in ATMEL AVR devices [11, 25, 44, 45, 47, 50, 51]. In general, these previous works present benchmarking results in devices from the AtMega family rather than the AtTiny one, hence tolerating larger code sizes and RAM use. As they are hardly comparable with ours and because of place constraints, we do not detail them in this section. Overall, we believe they provide a complementary view to ours. In particular, the pretty complete comparisons of the XBX website certainly sheds another light on the different algorithms [51]. Note also that some of these previous works consider older versions of the SHA3 candidates. Our following results consider the exact SHA3 finalists, according to their last updated specifications. We recall that for the functions appearing several times in the tables (e.g. Keccak, Skein, Quark, PHOTON, SPONGENT), the different lines correspond to different specifications and not different implementations of the same algorithm.

Following Section 2.3, we evaluated the performances of our different algorithms based on three main metrics, namely the code size (in bytes), RAM use (in bytes) and cycle counts for different message sizes[4]. They are represented in Figures 2.1, 2.2 and 2.3. Besides, we also produced so-called combined metrics that aim to summarize the efficiency of the hash functions in the AtTiny45. We used the product of the code size and cycle count and the product of the RAM use and cycle count for this purpose. Eventually, we additionally provide all our results in four complete tables in Appendix. As already mentioned, these results have to be interpreted with care, as they both represent the skills of the programmer and the algorithms efficiency. Yet, given this cautionary note, we believe a number of general observations can be extracted.

First, the code size and RAM use illustrate that the implementation constraints were reached for all algorithms. Nevertheless, the cost of the SHA3 candidates is generally higher than the one of both lightweight hash functions and block cipher based constructions. For some of them, the RAM use is close to the limit of the AtTiny device (i.e. 256). This can be explained by the generally larger states of all SHA3 candidates.

Second, we observe that lightweight algorithms have large cycle counts compared to other hash functions. This implies that their overall efficiency (measured with the combined metrics) is generally low in our implementation context. By contrast, the flexible nature of sponge-based functions (including all lightweight proposals) allows reducing the RAM use quite significantly, which is an interesting feature for hardware and embedded software implementations.

Third, it is noticeable that the SHA3 candidates hardly compete with AES-256 in Hirose construction or Rijndael-256-256 in Davies-Meyer mode. This observation is quite consistently observed for all our metrics.

Eventually, and as far as SHA3 finalists (in the 256-bit versions) are concerned, our investigations suggest that BLAKE offers the best performance figures, followed by Grøstl, Keccak, Skein and JH.

All these results were naturally obtained within a limited time frame. Hence, we encourage the reader to download codes and possibly improve them with further optimization. Looking at how the AES implementations have evolved following its selection as standard, it is likely that similar improvements can be expected for the hash functions in this work.

Acknowledgements. This work has been funded in part by the European Commission's ECRYPT-II NoE (ICT-2007-216676), by the Belgian State's IAP program P6/26 BCRYPT, by the ERC project 280141 (acronym CRASH), by the 7th framework European project TAMPRES, by the Walloon region's S@T Skywin and MIPSs projects. Stéphanie Kerckhof is a PhD student funded by a FRIA grant, Belgium. François-Xavier Standaert is a Research Associate of the Belgian Fund for Scientific Research (FNRS-F.R.S). Zheng Gong is supported by NSFC (No. 61100201).

[4] Note that for certain (e.g. sponge-based) functions, the data part of the RAM could be arbitrarily reduced by changing the interface. In this case, the RAM use evaluation in the figures excluded the data RAM (reported in gray in the tables).

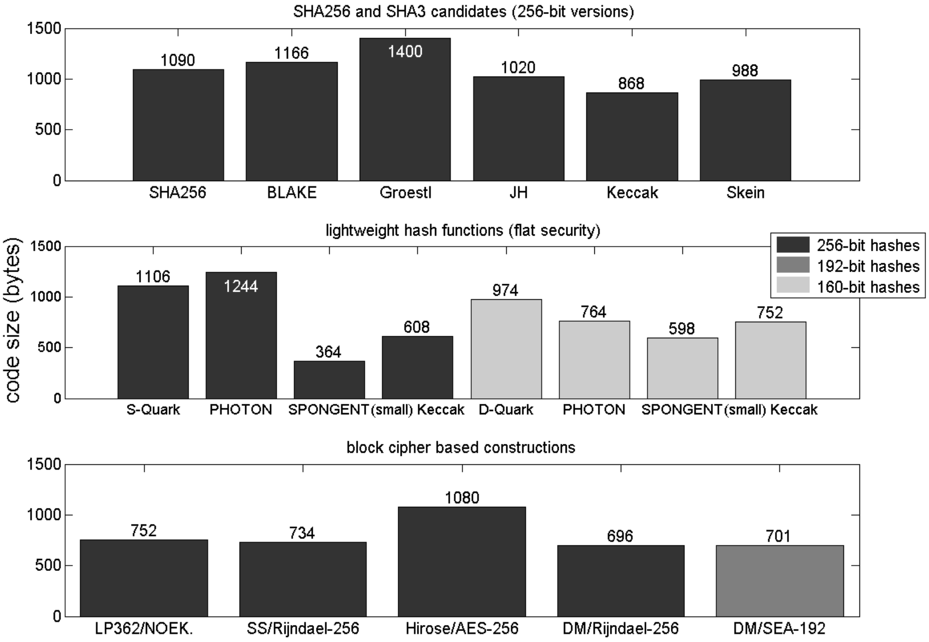

Figure 2.1: Performance evaluation: code size (bytes).

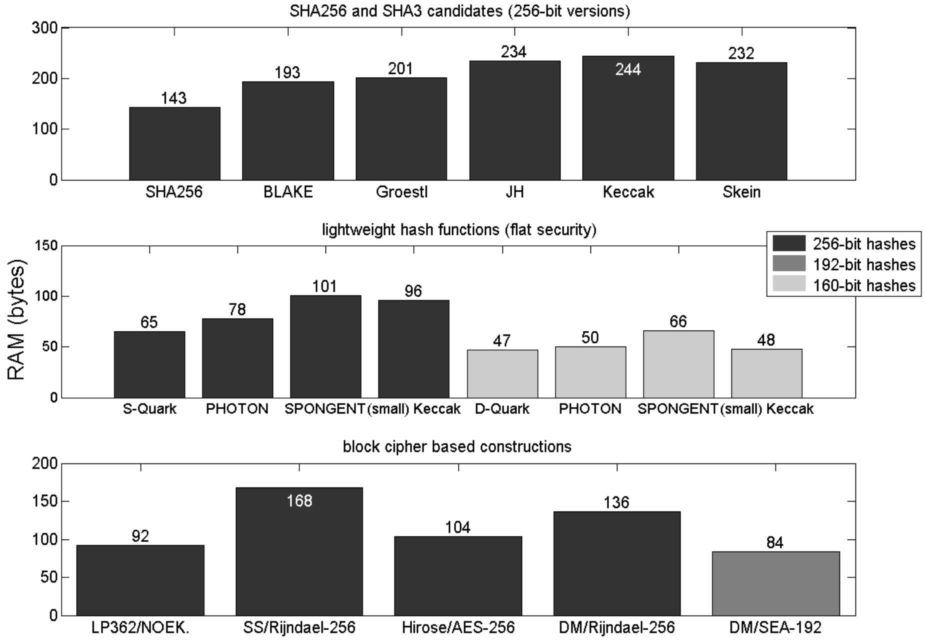

Figure 2.2: Performance evaluation: RAM (bytes).

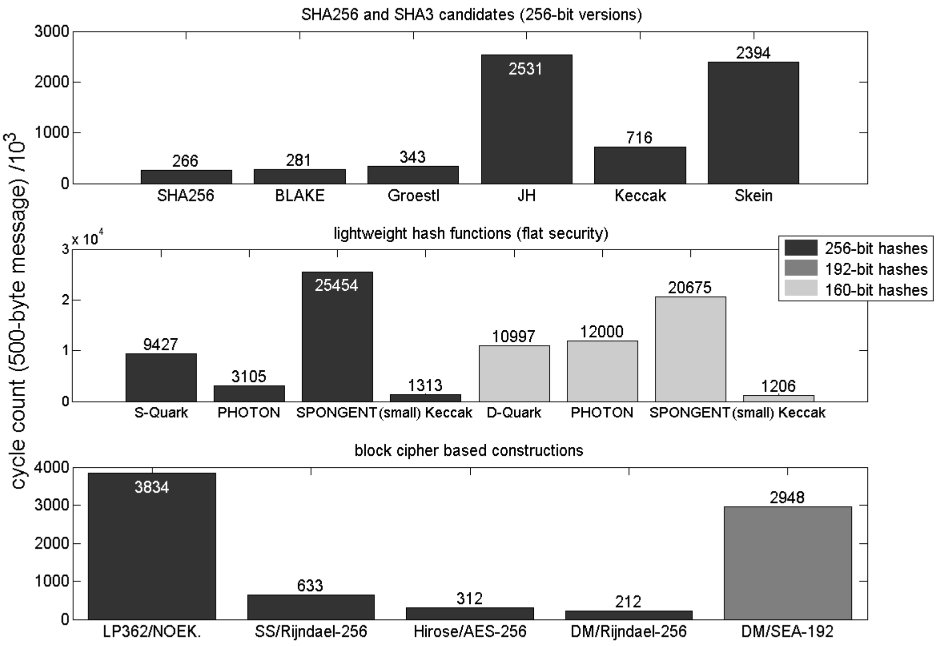

Figure 2.3: Performance evaluation: cycle count (500-byte message).

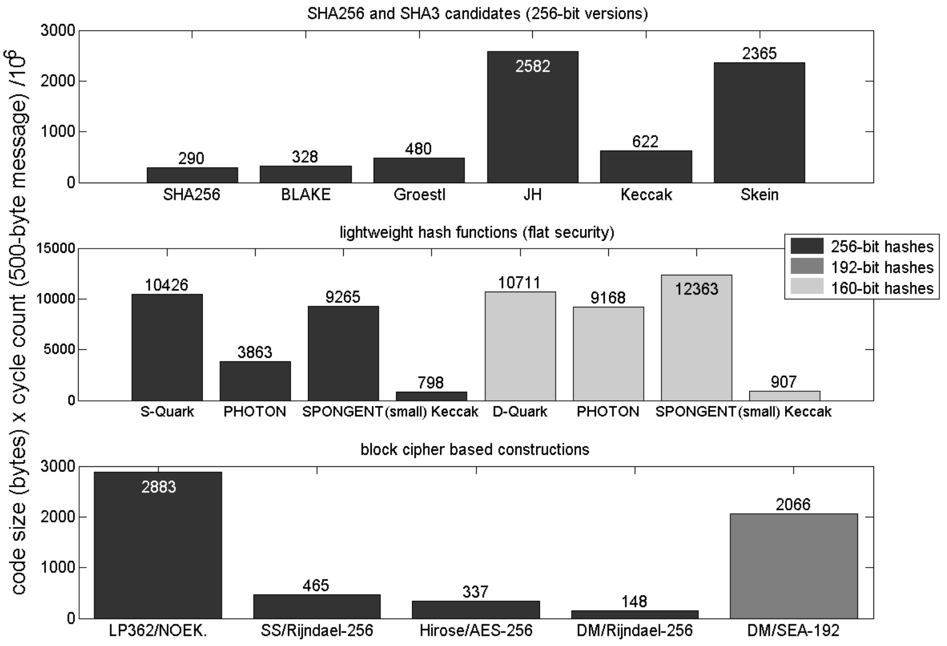

Figure 2.4: Performance evaluation: code size (bytes) x cycle count (500-byte message).

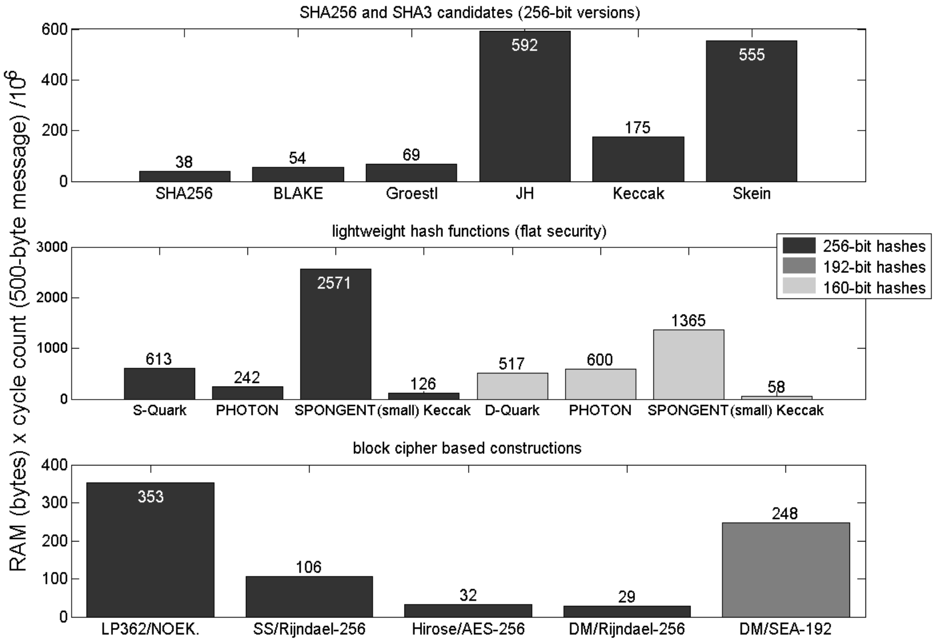

Figure 2.5: Performance evaluation: RAM (bytes) x cycle count (500-byte message).

Table 1: SHA256 and SHA3 candidates.

Hash function	**digest size**	**code size**	**RAM** [bytes]			**cycle count(message length)**			
	[bits]	[bytes]	**data**	**state**	**stack**	(8-byte)	(50-byte)	(100-byte)	(500-byte)
SHA256	256	1090	64	73	6	33 600	33 600	66 815	266 105
BLAKE-256	256	1166	64	120	9	35 714	35 714	70 808	281 372
Groestl-256	256	1400	64	128	9	61 007	61 049	101 279	342 759
JH-256	256	1020	64	162	8	524 602	524 518	785 510	2 531 262
Keccak[r=1088,c=512]*	256	868	gris25*136*	*240*	*4*	178 022	178 022	179 494	716 483
Skein-512-256	256	988	64	160	8	532 346	532 388	798 290	2 393 802

* benchmarked on the AtTiny85 because the read-only input buffer did not fit in the ATtiny45 and our interface did not allow that the message could be directly XORed into the state, hence resulting in an excessive RAM for the AtTiny45.

Table 2: Lightweight hash functions.

Hash function	**digest size**	**code size**	**RAM** [bytes]			**cycle count(message length)**			
	[bits]	[bytes]	**data**	**state**	**stack**	(8-byte)	(50-byte)	(100-byte)	(500-byte)
S-Quark	256	1106	gris254	60	5	708 783	1 417 611	2 339 023	9 4270 23
PHOTON-256/32/32	256	1244	gris254	68	10	254 871	486 629	787 896	3 105 396
SPONGENT-256/256/128	256	364	gris2516	96	5	1 542 923	3 856 916	6 170 900	25 454 100
Keccak[r=144,c=256]	256	608	gris2518	92	4	90 824	181 466	317 221	1 313 291
D-Quark	160	974	gris252	42	5	631 871	1 516 685	2 570 035	10 996 835
PHOTON-160/36/36	160	764	gris259	39	11	620 921	1 655 364	2 793 265	11 999 914
SPONGENT-160/160/80	160	598	gris2510	60	6	795 294	2 783 241	4 771 186	20 674 746
Keccak[r=40,c=160]	160	752	gris255	45	3	58 063	162 347	278 269	1 205 627

Table 3: Block cipher based constructions.

Hash function	**digest size**	**code size**	**RAM** [bytes]			**cycle count(message length)**			
	[bits]	[bytes]	**data**	**state**	**stack**	(8-byte)	(50-byte)	(100-byte)	(500-byte)
LP362/NOEK.	256	752	16	66	10	239 828	479 485	838 884	3 833 859
SS/Rijndael-256	256	734	32	130	6	39 738	79 313	158 429	633 225
Hirose/AES-256	256	1080	16	82	6	9918	39 111	68 304	311 579
DM/Rijndael-256	256	696	32	98	6	13 438	26 705	53 205	212 305
SM/SEA-192	192	701	24	50	10	134 121	402 055	669 997	2 947 536

Table 4: Additional results.

Hash function	**digest size**	**code size**	**RAM** [bytes]			**cycle count(message length)**			
	[bits]	[bytes]	**data**	**state**	**stack**	(8-byte)	(50-byte)	(100-byte)	(500-byte)
SHA256 (fast)	256	1242	64	73	6	26 208	26 208	52 031	206 969
Keccak[r=544,c=256]	256	672	gris2568	120	4	93 170	93 842	187 153	748 619
Keccak[r=640,c=160]	160	672	gris2580	120	3	93 170	93 842	187 153	656 108
Keccak[r=240,c=160]	160	570	gris2530	60	3	45 394	91 051	181 821	773 026
Skein-256-256	256	1316	32	80	10	212 872	319 154	531 681	1 806 949
lp362/NOEK.	256	650	16	66	10	239 200	478 233	836 696	3 823 871

Bibliography

[1] http://perso.uclouvain.be/fstandae/lightweight_ciphers/.

[2] http://point-at-infinity.org/avraes/.

[3] 3rd Generation Partnership Project. Technical specification group services and system aspects, 3g security, specification of the 3gpp confidentiality and integrity algorithms, document 2: Kasumi specification (release 10), 2011.

[4] ATMEL. Avr 8-bit microcontrollers, http://www.atmel.com/products/avr/.

[5] J.-P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia. Quark: A lightweight hash. In Mangard and Standaert [38], pages 1–15.

[6] J.-P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia. QUARK C implementation. Available at https://www.131002.net/quark/, 2010.

[7] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST (Round 3), 2010.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. Ecrypt Hash Workshop 2007, May 2007. also available as public comment to NIST from http://www.csrc.nist.gov/pki/HashWorkshop/Public$_Comments/2007_May.html$.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference, January 2011. http://keccak.noekeon.org/.

[10] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission, January 2011. http://keccak.noekeon.org/.

[11] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. Keccak implementation overview, September 2011. http://keccak.noekeon.org/.

[12] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. Spongent: The design space of lightweight cryptographic hashing. *IACR Cryptology ePrint Archive*, 2011:697, 2011.

[13] 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 P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007.

[14] G. Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[15] C. D. Cannière, O. Dunkelman, and M. Knezevic. Katan and ktantan - a family of small and efficient hardware-oriented block ciphers. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 272–288. Springer, 2009.

[16] J. Daemen, M. Peeters, G. V. Assche, and V. Rijmen. Nessie proposal: NOEKEON, 2000. Available online at http://gro.noekeon.org/Noekeon-spec.pdf.

[17] J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen. Nessie proposal: NOEKEON, 2000. Available online at http://gro.noekeon.org/Noekeon-spec.pdf.

[18] J. Daemen and V. Rijmen. The block cipher rijndael. In J.-J. Quisquater and B. Schneier, editors, *CARDIS*, volume 1820 of *Lecture Notes in Computer Science*, pages 277–284. Springer, 1998.

[19] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[20] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[21] J. Daemen and V. Rijmen. AES proposal: Rijndael. In *Proc. first AES conference*, August 1998. Available on-line from the official AES page: `http://csrc.nist.gov/encryption/aes/aes_home.htm`.

[22] I. Damgård. A design principle for hash functions. In Brassard [14], pages 416–427.

[23] G. de Meulenaer, F. Gosset, F.-X. Standaert, and O. Pereira. On the energy cost of communication and cryptography in wireless sensor networks. In *WiMob*, pages 580–585. IEEE, 2008.

[24] 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, *AFRICACRYPT*, volume 7374 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2012.

[25] T. Eisenbarth, S. Heyse, I. von Maurich, T. Poeppelmann, J. Rave, C. Reuber, and A. Wild. Evaluation of sha-3 candidates for 8-bit embedded processors. The Second SHA-3 Candidate Conference, 2010.

[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, 2007.

[27] J. Feichtner. http://www.groestl.info/implementations.html.

[28] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The skein hash function family (version 1.3), 2010. http://www.skein-hash.info/.

[29] K. Gaj, E. Homsirikamol, and M. Rogawski. Fair and comprehensive methodology for comparing hardware performance of fourteen round two sha-3 candidates using fpgas. In Mangard and Standaert [38], pages 264–278.

[30] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Sha-3 proposal grøstl (version 2.0.1), 2011. http://www.groestl.info/.

[31] J. Guo, T. Peyrin, and A. Poschmann. The PHOTON family of lightweight hash functions. In P. Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.

[32] S. Hirose. Some plausible constructions of double-block-length hash functions. In M. J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2006.

[33] 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, *CHES*, volume 4249 of *LNCS*, pages 46–59. Springer, 2006.

[34] X. Lai and J. L. Massey. A proposal for a new block encryption standard. In *EUROCRYPT*, pages 389–404, 1990.

[35] G. Leander, C. Paar, A. Poschmann, and K. Schramm. New lightweight des variants. In A. Biryukov, editor, *FSE*, volume 4593 of *LNCS*, pages 196–210. Springer, 2007.

[36] J. Lee and J. H. Park. Preimage resistance of lpmkr with r=m-1. *Inf. Process. Lett.*, 110(14-15):602–608, 2010.

[37] C. H. Lim and T. Korkishko. mcrypton - a lightweight block cipher for security of low-cost rfid tags and sensors. In J. Song, T. Kwon, and M. Yung, editors, *WISA*, volume 3786 of *LNCS*, pages 243–258. Springer, 2005.

[38] S. Mangard and F.-X. Standaert, editors. *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *LNCS*. Springer, 2010.

[39] M. Matsui. New block encryption algorithm misty. In E. Biham, editor, *FSE*, volume 1267 of *LNCS*, pages 54–68. Springer, 1997.

[40] R. C. Merkle. One way hash functions and des. In Brassard [14], pages 428–446.

[41] National Institute of Standards and Technology. FIPS 180-3, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-3. Technical report, U.S. Department of Commerce, Oct. 2008.

[42] NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register Notices*, 72(212):62212–62220, November 2007. http://csrc.nist.gov/groups/ST/hash/index.html.

[43] NIST. NIST special publication 800-57, recommendation for key management (revised), March 2007.

[44] D. A. Osvik. Fast embedded software hashing. Cryptology ePrint Archive, Report 2012/156, 2012. http://eprint.iacr.org/.

[45] D. Otte. Avr-crypto-lib, 2009. http://www.das-labor.org/wiki/Crypto-avr-lib/en.

[46] P. Rogaway and J. P. Steinberger. Constructing cryptographic hash functions from fixed-key blockciphers. In D. Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 433–450. Springer, 2008.

[47] G. Roland. Efficient implementation of the grøstl-256 hash function on an atmega163 microcontroller. Available at http://groestl.info/groestl-0-8bit.pdf, June 2009.

[48] T. Shrimpton and M. Stam. Building a collision-resistant compression function from non-compressing primitives. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 643–654. Springer, 2008.

[49] F.-X. Standaert, G. Piret, N. Gershenfeld, and J.-J. Quisquater. Sea: A scalable encryption algorithm for small embedded applications. In J. Domingo-Ferrer, J. Posegga, and D. Schreckling, editors, *CARDIS*, volume 3928 of *LNCS*, pages 222–236. Springer, 2006.

[50] J. Walter. Fhreefish (skein implementation) website. http://www.syntax-k.de/projekte/fhreefish/.

[51] C. Wenzel-Benner, J. Gräf, J. Pham, and J.-P. Kaps. XBX benchmarking results january 2012. Third SHA-3 candidate conference, http://xbx.das-labor.org/trac/wiki/r2012platforms$_atmega1284p_16mhz, Mar2012$.

[52] D. J. Wheeler and R. M. Needham. Tea, a tiny encryption algorithm. In B. Preneel, editor, *FSE*, volume 1008 of *LNCS*, pages 363–366. Springer, 1994.

[53] H. Wu. JH Documentation Website. http://www3.ntu.edu.sg/home/wuhj/research/jh/.

[54] H. Wu. The Hash Function JH, January 2011. http://www3.ntu.edu.sg/home/wuhj/research/jh/.