Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices

Josep Balasch[1], Barış Ege[2], Thomas Eisenbarth[3], Benoit Gérard[4], Zheng Gong[5], Tim Güneysu[6], Stefan Heyse[6], Stéphanie Kerckhof[4], François Koeune[4], Thomas Plos[7], Thomas Pöppelmann[6], Francesco Regazzoni[8], François-Xavier Standaert[4], Gilles Van Assche[9], Ronny Van Keer[9], Loïc van Oldeneel tot Oldenzeel[4], Ingo von Maurich[6].

[1] Department of Electrical Engineering ESAT/COSIC, KULeuven, Belgium.
[2] Digital Security Group - ICIS, Radboud Universiteit Nijmegen, The Netherlands.
[3] Dept. of Electrical & Computer Engineering, Worcester Polytechnic Institute, USA.
[4] ICTEAM/ELEN/Crypto Group, Université catholique de Louvain, Belgium.
[5] School of Computer Science, South China Normal University.
[6] Horst Görtz Institute for IT-Security, Ruhr-Universität Bochum, Germany.
[7] Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Austria.
[8] ALaRI Institute, University of Lugano, Switzerland.
[9] STMicroelectronics.

Abstract. The pervasive diffusion of electronic devices in security and privacy sensitive applications has boosted research in cryptography. In this context, the study of lightweight algorithms has been a very active direction over the last years. In general, symmetric cryptographic primitives are good candidates for low-cost implementations. For example, several previous works have investigated the performance of block ciphers on various platforms. Motivated by the recent SHA3 competition, this paper extends these studies to another family of cryptographic primitives, namely hash functions. We implemented different algorithms on an ATMEL AVR ATtiny45 8-bit microcontroller, and provide their performance evaluation. All the implementations were carried out with the goal of minimizing the code size and memory utilization, and are evaluated using a common interface. As part of our contribution, we make all the corresponding source codes available on a web page, under an open-source license. We hope that this paper provides a good basis for researchers and embedded system designers who need to include more and more functionalities in next generation smart devices.

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 performance 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 [14]. In order to increase the relevance of their work, the authors additionally provided open source codes for all their implementations on a public web page. 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 [23], 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] [24]. 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). 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 provide all our source codes on a public web page [1]. As a result, we hope that this initiative can be used as a first step in better understanding the performance of hash functions in a specific but meaningful class of devices.

Selected algorithms. We investigated hash functions in three main categories. First, we considered SHA256 [22] and SHA3 candidates BLAKE-256 [4], Grøstl-256 [18], JH-256 [34], Keccak[r=1088,c=512] [6, 7] and Skein-512-256 [17]. Second, we evaluated the lightweight hash functions Quark (S and Q versions) [2], PHOTON (160/36/36 and 256/32/32 versions) [19], SPONGENT (160/160/80 and 256/256/128 versions) [9] and Keccak (i.e. low-cost alternatives to the standard version). Eventually, we also focused on block cipher based constructions such as Rogaway-Steingberger [27], Hirose [20], Davies-Meyer and Shrimpton-Stam [29], based on NOEKEON [10], AES-256, Rijndael 256 [11] and SEA-192 [30]. More details on these algorithms are given in the extended paper [5].

2 Methodology and Metrics

In order to be able to compare the performance 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

[1] i.e. the same security is required for collision, preimage and 2nd preimage resistance.

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. (a) *init* takes no input and initializes the internal state, which is a dedicated memory zone seen as a black box, and returns no output; (b) *update* takes as input a full block of data, updates its internal state by processing that block and returns no output; (c) *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 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. 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. Finally note that some of the guidelines were not always followed, because of the cipher specifications making them less relevant (which will be specified when necessary).

3 Description of the ATtiny45 Microcontroller

The ATtiny45 is a 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 timers, analog-to-digital converters or serial interfaces. 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 integrates 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. The ATtiny45 also has lower power consumption and is cheaper.

4 Implementation Details

4.1 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. When implementing the iteration step of its compression function, the main observation is that six out of eight working registers are just circularly copied. To reduce code and cycles for memory transfer operations, the addresses of the RAM-based working registers are reassigned using circular pointer arithmetic instead of addressing these registers by its names `A-H` explicitly.

Circular pointer arithmetic as part of the iteration step is also 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 instead of 160 bytes in ROM. In order to maintain a decent performance while keeping the code size down we incorporated the observation by Osvik [25] to efficiently load and store in-/outputs of the round function $G_i\,(a, b, c, d)$. Furthermore, we use loops where applicable and move recurring tasks 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 performance 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 [16, 28], 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 require to store 42 256-bit round constants in memory, which is not compliant with our low-cost constraints. Hence, JH was implemented according to the reference specifications. The utilization 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 performance, 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 [33]. For the same reason, the initial state was precomputed and stored in program memory. It allows us to save the initialization phase which is equivalent to the processing of one input block. Regarding the permutation, it is performed by reading the state bytes in a different order at the beginning of each round. Finally, the state bits are reorganized at the beginning 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. 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 [8]. 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 employed. 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 performance, by leaving most of the time three out of the four state words in the registers.

4.2 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 performance. 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 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

[2] During implementation, a minor inconsistency was discovered between the paper description [2] and the reference code [3], 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.

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 LUT [12].

SPONGENT-160/160/80 and -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.

4.3 Block Cipher-based Constructions

Rogaway-Steinberger LP/lp362. For realizing the Rogaway and Steinberger construction principle, the matrix A suggested by Lee and Park [21] with $\alpha = 2$ has been used. For operations in $\mathbb{F}_{2^{128}}$ (addition and multiplication) we have selected the same irreducible polynomial $x^{128} + x^7 + x^2 + x + 1$ as stated in [27]. The implementation of the block cipher NOEKEON is based on the open source version published in [14], 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 is similar to the one presented in [14], 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 [13]. 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 [14]. 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.

5 Performance Evaluation and Conclusions

We first refer to a number of other implementations of hash functions in ATMEL AVR devices [8, 15, 25, 26, 28, 31, 32]. 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 space 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 [32]. 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, we evaluated the performance 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 1, 2 and 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 note that all our results are given numerically in the performance tables of the extended paper [5]. 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 usage 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 usage 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.

Acknowledgments. 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).

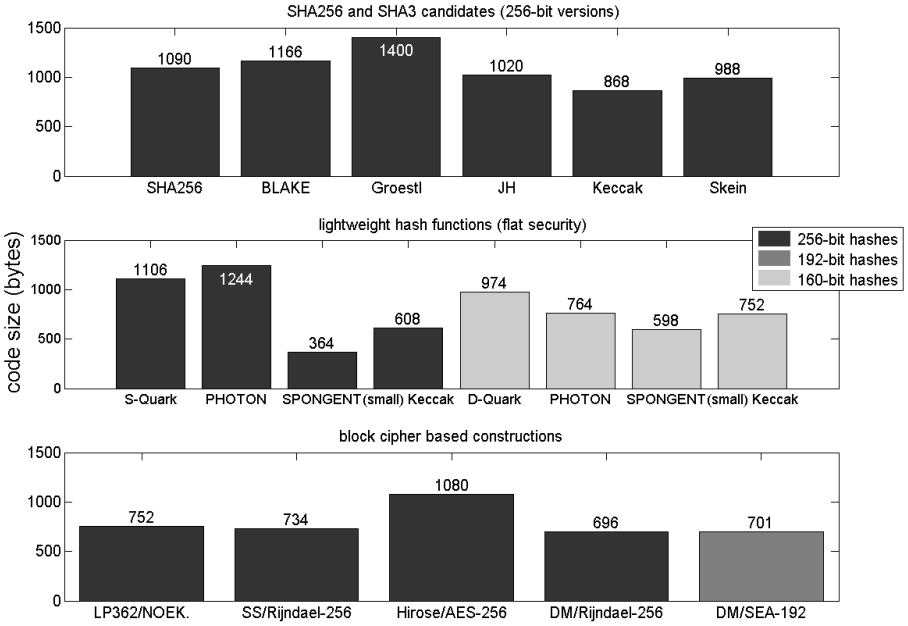

Fig. 1: Performance evaluation: code size (bytes).

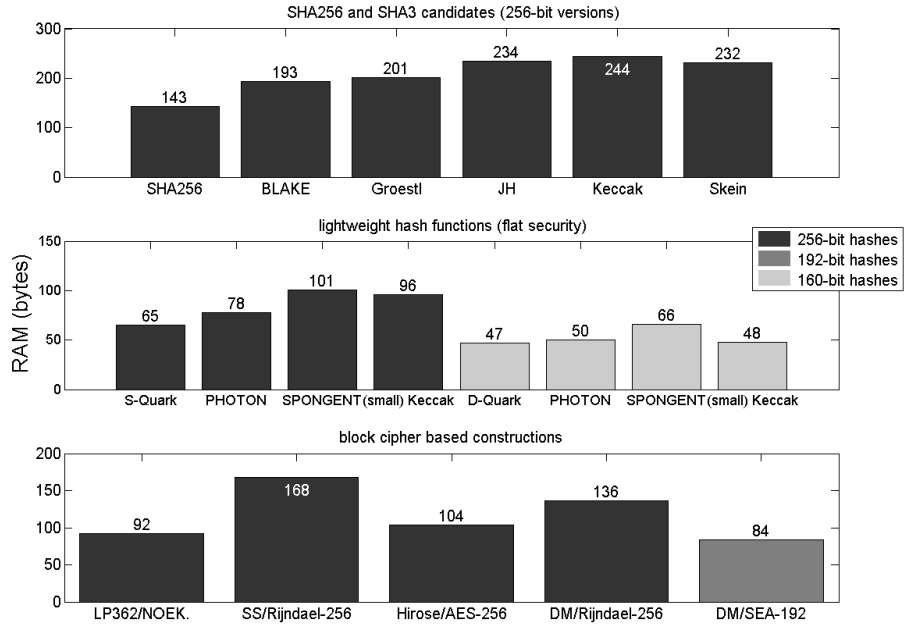

Fig. 2: Performance evaluation: RAM (bytes).

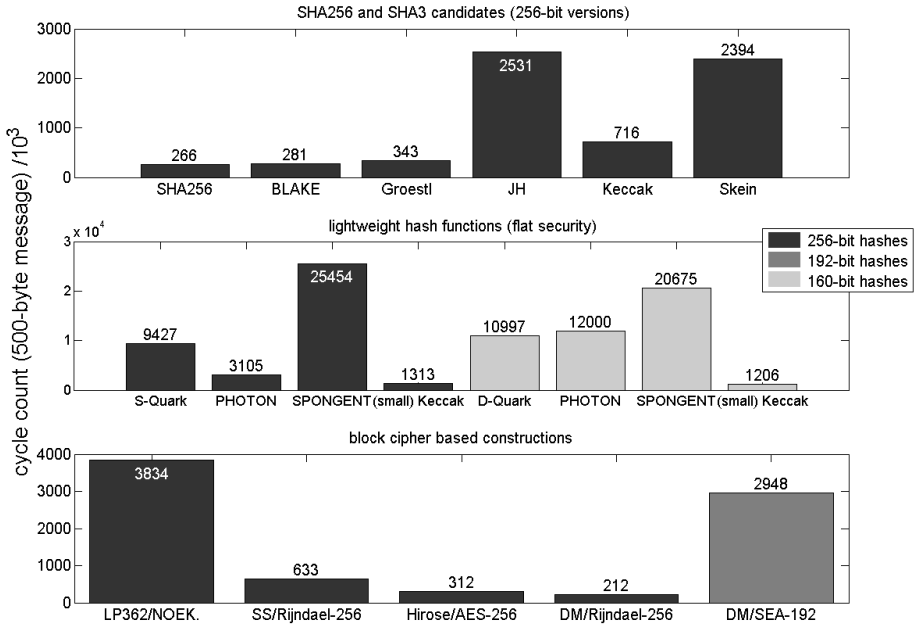

Fig. 3: Performance evaluation: cycle count (500-byte message).

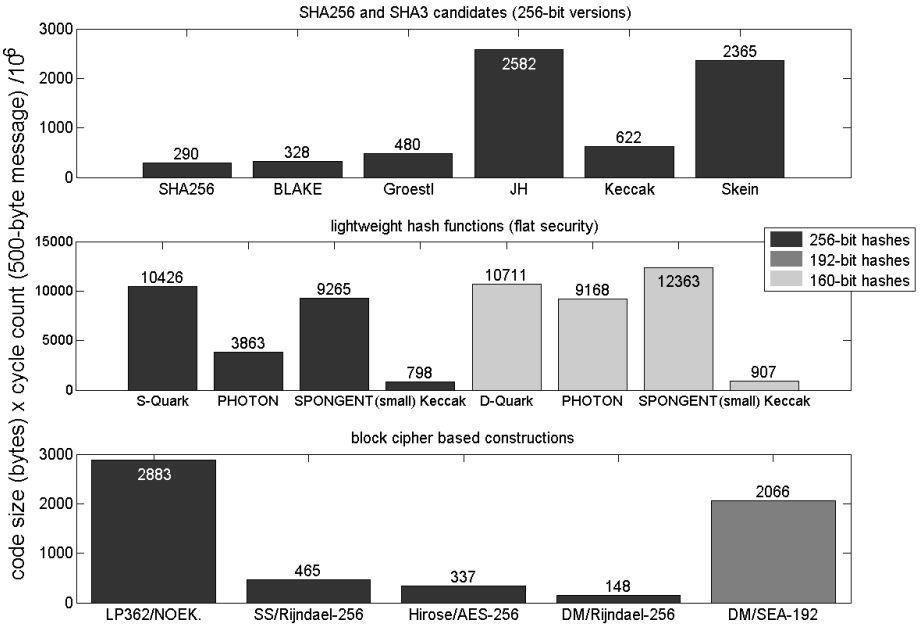

Fig. 4: Performance evaluation: code size (bytes) x cycle count (500-byte message).

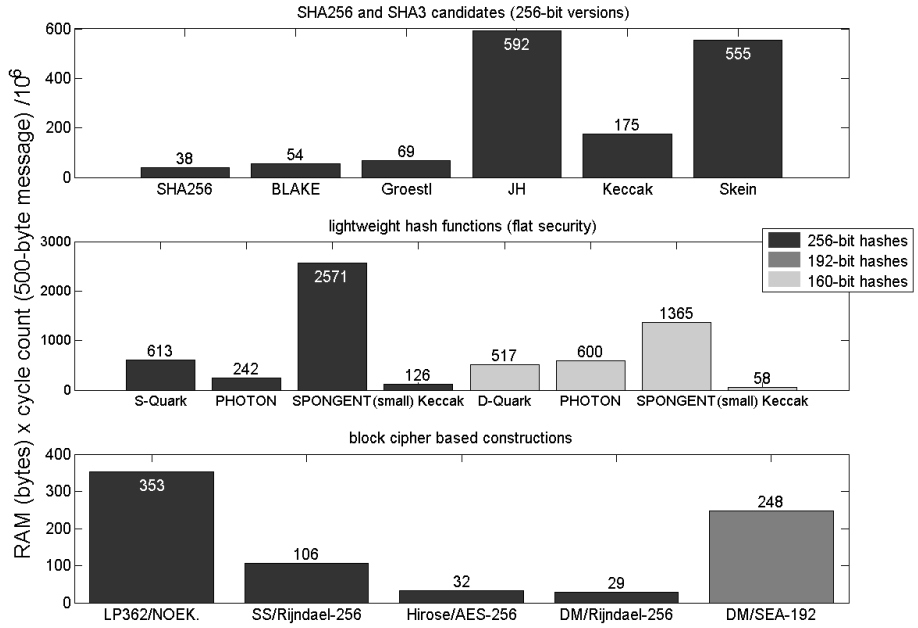

Fig. 5: Performance evaluation: RAM (bytes) x cycle count (500-byte message).

References

1. `http://perso.uclouvain.be/fstandae/source_codes/hash_atmel/`.
2. J.-P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia. Quark: A lightweight hash. In S. Mangard and F.-X. Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010.
3. J.-P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia. QUARK C implementation. Available at https://www.131002.net/quark/, 2010.
4. J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST (Round 3), 2010.
5. J. Balasch, B. Ege, T. Eisenbarth, B. Gérard, Z. Gong, T. Güneysu, S. Heyse, S. Kerckhof, F. Koeune, T. Plos, T. Pöppelmann, F. Regazzoni, F.-X. Standaert, G. V. Assche, R. V. Keer, L. van Oldeneel tot Oldenzeel, and I. von Maurich. Compact implementation and performance evaluation of hash functions in attiny devices. Cryptology ePrint Archive, Report 2012/507, 2012. `http://eprint.iacr.org/`.
6. 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`.
7. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference, January 2011. `http://keccak.noekeon.org/`.
8. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. Keccak implementation overview, September 2011. `http://keccak.noekeon.org/`.

9. 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.
10. J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen. Nessie proposal: NOEKEON, 2000. Available online at `http://gro.noekeon.org/Noekeon-spec.pdf`.
11. 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.
12. J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
13. 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`.
14. 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.
15. 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.
16. J. Feichtner. http://www.groestl.info/implementations.html.
17. 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/`.
18. 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/`.
19. 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.
20. 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.
21. J. Lee and J. H. Park. Preimage resistance of lpmkr with r=m-1. *Inf. Process. Lett.*, 110(14-15):602–608, 2010.
22. 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.
23. 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`.
24. NIST. NIST special publication 800-57, recommendation for key management (revised), March 2007.
25. D. A. Osvik. Fast embedded software hashing. Cryptology ePrint Archive, Report 2012/156, 2012. `http://eprint.iacr.org/`.
26. D. Otte. Avr-crypto-lib, 2009. `http://www.das-labor.org/wiki/Crypto-avr-lib/en`.

27. 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.
28. 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.
29. 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.
30. 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 *Lecture Notes in Computer Science*, pages 222–236. Springer, 2006.
31. J. Walter. Fhreefish (skein implementation) website. `http://www.syntax-k.de/projekte/fhreefish/`.
32. 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`, Mar 2012.
33. H. Wu. JH Documentation Website. `http://www3.ntu.edu.sg/home/wuhj/research/jh/`.
34. H. Wu. The Hash Function JH, January 2011. `http://www3.ntu.edu.sg/home/wuhj/research/jh/`.