Compact Implementation and Performance Evaluation of Block Ciphers in ATtiny Devices

Thomas Eisenbarth[1], Zheng Gong[2], Tim Güneysu[3], Stefan Heyse[3],
Sebastiaan Indesteege[4,5], Stéphanie Kerckhof[6], François Koeune[6],
Tomislav Nad[7], Thomas Plos[7], Francesco Regazzoni[6,8],
François-Xavier Standaert[6], Loic van Oldeneel tot Oldenzeel[6].

[1] Department of Mathematical Sciences, Florida Atlantic University, FL, USA.
[2] School of Computer Science, South China Normal University.
[3] Horst Görtz Institute for IT Security, Ruhr-Universität, Bochum, Germany.
[4] Department of Electrical Engineering ESAT/COSIC, KULeuven, Belgium.
[5] Interdisciplinary Institute for BroadBand Technology (IBBT), Ghent, Belgium.
[6] UCL Crypto Group, Université catholique de Louvain, Belgium.
[7] Institute for Applied Information Processing and
Communications (IAIK), Graz University of Technology, Austria.
[8] ALaRI Institute, University of Lugano, Switzerland.

Abstract. The design of lightweight block ciphers has been a very active research topic over the last years. However, the lack of comparative source codes generally makes it hard to evaluate the extent to which different ciphers actually reach their low-cost goals, on different platforms. This paper reports on an initiative aimed to partially relax this issue. First, we implemented 12 block ciphers on an ATMEL ATtiny45 device, and made the corresponding source code available on a webpage, with an open-source license. Common design goals and interface have been sent to all designers in order to enhance the comparability of the implementation results. Second, we evaluated the performances of these implementations according to different metrics, including energy-consumption measurements. Although inherently limited by slightly different design choices, we hope this initiative can trigger more work in this direction, e.g. by extending the list of implemented ciphers, or adding countermeasures against physical attacks in the future.

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], the XBX frameworks from CHES 2010 [20] and an interesting survey of lightweight cryptography implementations [10]. Unfortunately, these references are still limited by the number of ciphers under investigation and the fact that their source code is not always available for evaluation.

Following, the goal of our work is to extend the benchmarking of 12 lightweight and standard ciphers, and to make their implementation available under an open-source license. The ciphers were chosen according to three criteria: all selected candidates should (a) give no indication of flawed security, (b) be freely usable without patent restrictions and (c) likely result in lightweight implementations with a footprint of less than 256 bytes of RAM and 4 KB of code size for a combined encryption and decryption function.

In order to make comparisons as meaningful as possible, we tried to adapt the guidelines proposed in [11] for the evaluation of hardware implementations to our software context. Yet, as the project was involving 12 different designers, we also acknowledge that some biases can appear in our conclusions, due to slightly different implementation choices. Hence, as usual for performance evaluations, looking at the source codes is essential in order to properly understand the reasons of different performance figures. Overall, we hope that this initiative can be used as a first step in better analyzing 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 attacks (e.g. based on faults or side-channel leakage) as another evaluation criteria.

The rest of the paper is structured as follows. Section 2 contains a brief specification of the implemented ciphers. Section 3 establishes our evaluation methodology and metrics, followed by Section 4 that gives details about the ATtiny45 microcontroller. Section 5 provides succinct descriptions and motivation of the implementation choices made by our 12 designers. Finally, our performance evaluations are in Section 6 and conclusions are drawn in Section 7. The webpage containing all our open-source codes is given here [1].

2 List of Investigated Ciphers

AES Rijndael [8] 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 [15] 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 as for the original DES.

HIGHT [13] 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 [14] 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 as 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 input and output. A major drawback of its construction is the inverse key schedule that requires the complex extended Euclidean algorithm during decryption. For efficient implementation, this complex key schedule needs to be precomputed and stored in memory.

KASUMI [3] is a block cipher derived from MISTY1 [18]. 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 input and output. 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 [6] 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. A 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.

KLEIN [12] is a family of lightweight software oriented block ciphers with 64-bit plaintexts and variable key length (64, 80 or 96 bits - our performance evaluations focus on the 80-bit version). It is primarily designed for software implementations in resource-constrained devices such as wireless sensors and RFID tags, but its hardware implementation can be compact as well. The structure of KLEIN is a typical Substitution-Permutation Network (SPN) with 12/16/20 rounds for KLEIN-64/80/96 respectively. One round transformation consists of four operations AddRoundKey, SubNibbles (4-bit involutive S-box), RotateNibbles and MixNibbles (borrowed from AES MixColumns). The key schedule of KLEIN has a Feistel-like structure. It is agile even if keys are frequently changed and is designed to avoid potential related-key attacks.

mCrypton [16] is another block cipher 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 have 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 [7] is a block cipher with a key length and a block size of 128 bits. The block cipher consists of a simple round function based 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 [5] 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 or 128-bit key size (we focus on the 80-bit variant). PRESENT implements a substitution-permutation network and iterates 31 rounds. The permutation layer

consists only of bit permutations (i.e. wire crossings). 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 [19] 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 $SEA_{96,8}$, 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 [21] 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}.

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, detailed hereunder.

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 encryption process should start with plaintext and key in data memory. The ciphertext should overwrite the plaintext at the end of this process (and vice versa for decryption).
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 (also provided on [1]).

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 6). For the energy-consumption evaluations, each cipher has been flashed in an 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.

Note finally that, as mentioned in introduction, the 12 ciphers were implemented by 12 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. In particular, the following exceptions deserve to be mentioned. (1) The key scheduling of IDEA is not computed "on-the-fly" but precomputed (as explained in Section 2). (2) The key in KATAN has to be restored externally for subsequent invocations. (3) The 4-bit S-boxes of KLEIN, mCrypton, PRESENT were implemented as 8-bit tables (because of a better memory vs. speed tradeoff).

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.

5 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 straightforward and is written according to the specification. Beside the storage for plain/ciphertext and the keys k, k_1, k_2, 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, no dedicated hardware multiplier unit is available. Multiplication was therefore emulated in software resulting in a data-dependent 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. Note that IDEA's multiplication is special and maps zero as any input to 2^{16} (which is equivalent to $-1 \bmod 2^{16}+1$). Therefore, whenever a zero is detected as input to the multiplication, our implementations returns the additive inverse of the other input, reduced modulo $2^{16}+1$.

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 rearrangements were 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. To avoid excessive register pressure, the in- 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. The fact that three rounds of KATAN can be run in parallel was not used in this implementation. Doing so would require more complicated shifting and masking to extract bits from the state, and thus significantly increase the code size, for little or no performance gain. 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. Keeping the master key in RAM would require 10 additional words (note that the KTANTAN key schedule does not modify the key, so it does not have this limitation). In order to implement the non-linear functions efficiently, addition instructions were used to compute several logical AND's and XOR's in parallel through carefully positioning the input bits and using masking to avoid undesired carry propagation.

KLEIN-80. Despite the goal of small memory footprint, the 4-bit involutive S-box is stored as an 8-bit table for saving clock cycles. As it can be used in both encryption and decryption, this corresponds to a natural tradeoff between code size and processing speed (a similar choice is made for mCrypton and PRESENT, see the next paragraphs). To save memory usage during processing, the MixNibbles step (borrowed from AES MixColumns) is implemented by a single function without using lookup tables. Overall, 29 registers are used during the computations. Among them, 8 registers correspond to the intermediate state, 10 to the key scheduling, 9 registers are used for temporary storage and two for the round counter.

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

[1] All six variants of the KATAN/KTANTAN family are supported via conditional assembly. Our performance evaluations only focus on the 64-bit version of KATAN.

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 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.

6 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 $\times$ cycle count product, normalized by the block size. The results for our different implementations are given in Figures 2, 3, 4, 5, 6, 7 (all given in appendix). 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 [10]. As illustrated in Figure 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 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, PRESENT and KLEIN. This can in part be explained by the hardware-oriented flavor of these ciphers (e.g. the use of bit permutations or manipulation of 4-bit nibbles is not optimal in 8-bit microcontrollers). 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 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, this metric 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 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 KATAN 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 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 6) is strongly correlated with the cycle count, confirming the experimental results in [9]. 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 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 and KLEIN 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.

7 Conclusion

This paper reported on an initiative to evaluate the performance of different standard and lightweight block ciphers on a low cost micro-controller. 12 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 performance in mind. Scopes for further research include the extension of this work towards more algorithms and the addition of countermeasures against physical attacks.

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, MIPSs and NANOTIC-COSMOS projects. Stéphanie Kerckhof is a PhD student funded by a FRIA grant, Belgium. F.-X. Standaert is a Research Associate of the Belgian Fund for Scientific Research (FNRS-F.R.S). Zheng Gong is supported by NSFC (No. 61100201). The authors would like to thank Svetla Nikova for her help regarding the implementation of the block cipher KLEIN.

References

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. 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.
6. 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.

7. J. Daemen, M. Peeters, G. V. Assche, and V. Rijmen. Nessie proposal: NOEKEON, 2000. Available online at `http://gro.noekeon.org/Noekeon-spec.pdf`.
8. J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
9. 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.
10. 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.
11. 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 [17], pages 264–278.
12. Z. Gong, S. Nikova, and Y.-W. Law. Klein: A new family of lightweight block ciphers. to appear in the proceedings of RFIDsec 2011.
13. 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.
14. X. Lai and J. L. Massey. A proposal for a new block encryption standard. In *EUROCRYPT*, pages 389–404, 1990.
15. 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.
16. 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.
17. 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.
18. M. Matsui. New block encryption algorithm misty. In E. Biham, editor, *FSE*, volume 1267 of *LNCS*, pages 54–68. Springer, 1997.
19. 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.
20. C. Wenzel-Benner and J. Gräf. Xbx: external benchmarking extension for the supercop crypto benchmarking framework. In Mangard and Standaert [17], pages 294–305.
21. 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.

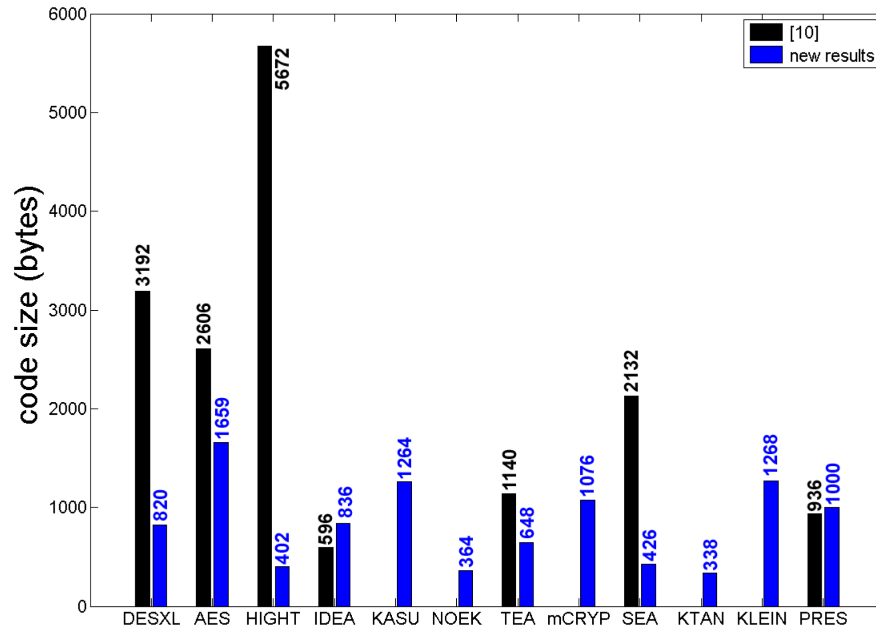

Fig. 1. Code size: comparison with previous work [10].

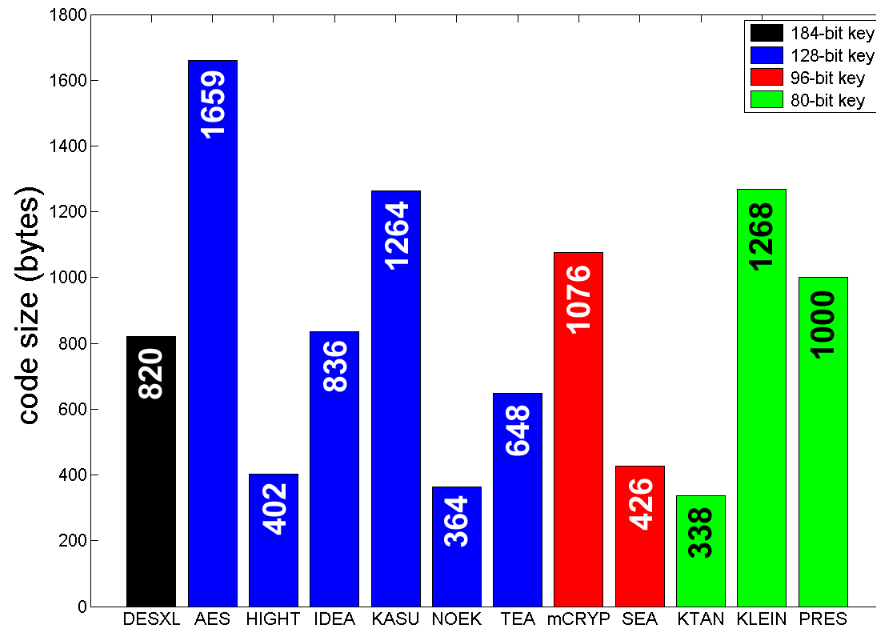

Fig. 2. Performance evaluation: code size.

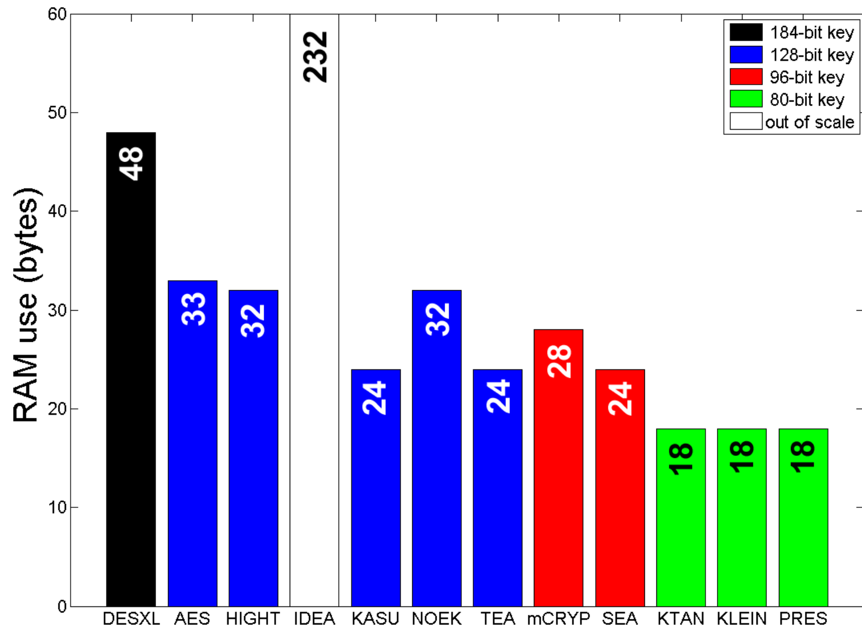

Fig. 3. Performance evaluation: RAM use.

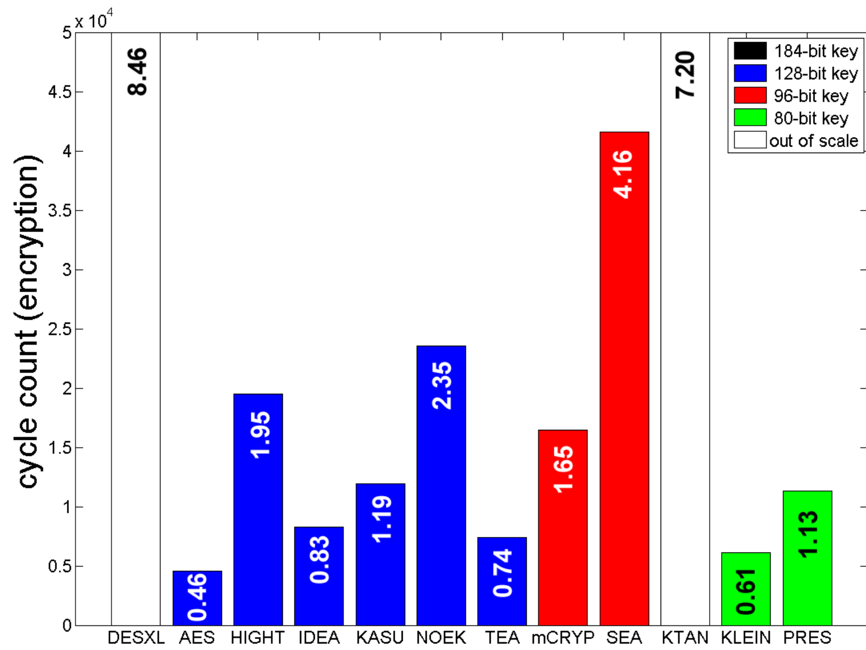

Fig. 4. Performance evaluation: cycle count (encryption).

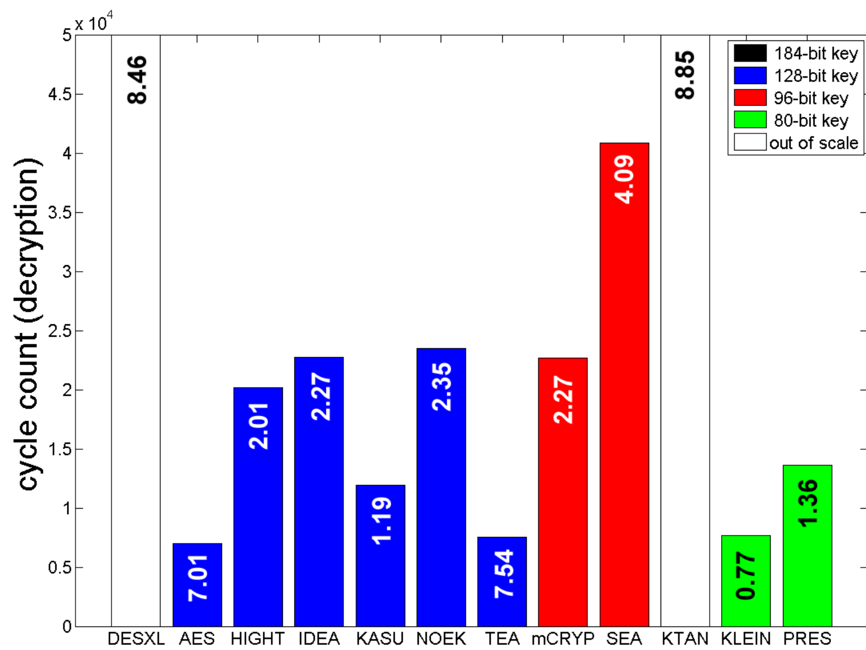

Fig. 5. Performance evaluation: cycle count (decryption).

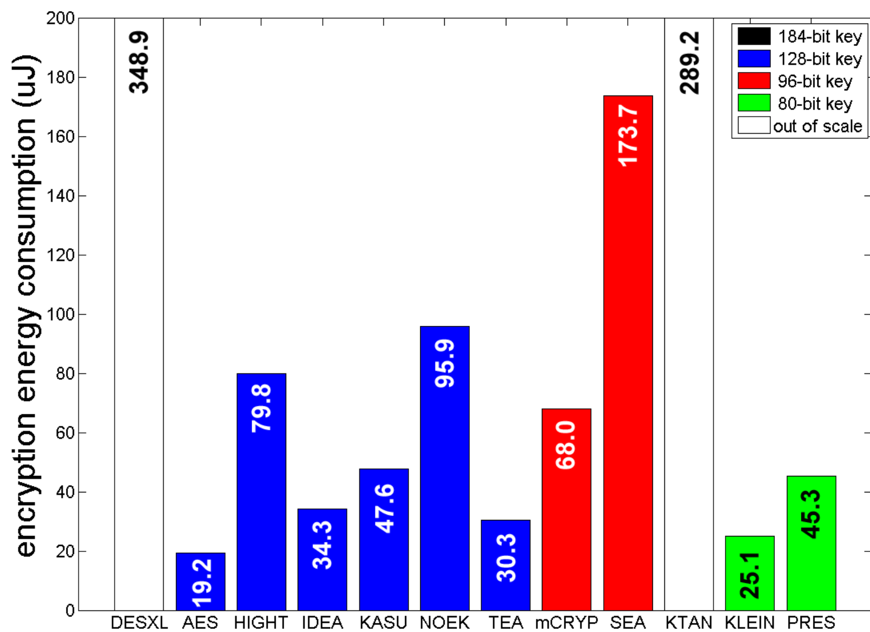

Fig. 6. Performance evaluation: energy consumption.

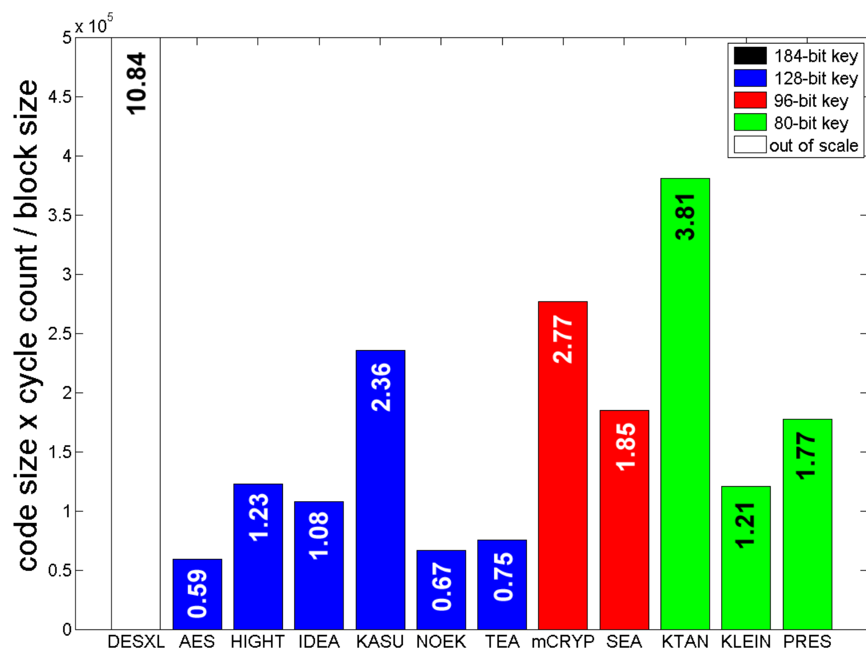

Fig. 7. Performance evaluation: combined metric.