Compact Implementation and Performance Evaluation of Block Ciphers in ATtiny Devices

No Author Given

No Institute Given

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 implementations of different ciphers actually reach their low-cost goals on various platforms. This paper reports on an initiative aiming to relax this issue. First, we provide implementations of 12 block ciphers on an ATMEL AVR ATtiny45 8-bit microcontroller, and make the corresponding source code available on a web page. All implementations are made public under an open-source license. Common interfaces and design goals are followed by all designers to achieve comparable implementation results. Second, we evaluate performance figures of our implementations with respect to different metrics, including energy-consumption measurements and show our improvements compared to existing implementations.

Keywords: Lightweight, Block Cipher, AVR ATtiny, Implementation, Open Source.

1 Introduction

Small embedded devices including smart cards, RFIDs, and sensor nodes are deployed in many applications today. They are usually characterized by strong cost constraints. Yet, as they increasingly manipulate sensitive data, they 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) turn out to be annoyingly expensive in software. Even within a class of similar targets, 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 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 trade-off for these algorithms.

In this paper, we provide performance evaluations for low-cost block ciphers, and investigate their implementation on an ATMEL AVR ATtiny45 device [3], i.e. a small 8-bit microcontroller with limited memory and limited 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 AES [18], the XBX frameworks [20] and an interesting survey of lightweight cryptography implementations [9]. Unfortunately, these references are still limited by the number of ciphers under investigation and the fact that in some cases the source code is not available for evaluation.

The goal of our work is to extend the benchmarking of 12 lightweight and standard block ciphers, namely AES, DESXL, HIGHT, IDEA, KASUMI, KATAN, KLEIN, mCrypton, NOEKEON, PRESENT, SEA, TEA, and to make their implementation available under an open-source license. To the best of our knowledge, four of these algorithms (KASUMI, KLEIN, mCrypton, KATAN) are implemented for the first time on an 8-bit platform. We selected the ciphers 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 adapt the guidelines for evaluations of hardware implementations proposed in [10] to our software context. Yet, as the project involves 12 different designers, we also acknowledge that some biases can appear 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 starting point to further develop cryptographic libraries for embedded platforms and, in the long run, add security against physical attacks (e.g., based on faults or side-channel leakage) as another evaluation criteria.

The remainder of this paper is structured as follows. Section 2 contains a brief overview 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 the designers. Finally, performance evaluations are given in Section 6 and conclusions are drawn in Section 7. The web page containing all our open-source codes is available at [2].

2 Investigated Ciphers

AES [7] is the new encryption standard selected in 2002 replacing the former DES. It supports key sizes of 128, 192 or 256 bits, and its block size is 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 applications, the typical choice is to fix the key size to 128 bits.

DESL, DESX, and DESXL [14] are lightweight variants of the DES cipher with the main goal to minimize the gate count required in hardware implementations. In 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.

HIGHT [12] 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 composed of simple operations: XOR, additions mod 2^8 and bitwise rotations. Its key schedule consists of two algorithms: one generating whitening key bytes for initial and final transformations; the other one generating subkeys for the 32 rounds. Each subkey byte is the result of an addition mod 2^8 between a master key byte and a constant generated using a linear feedback shift register.

IDEA [13] 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 [1] is a block cipher derived from MISTY1 [17]. It is used as a keystream generator in UMTS, GSM, and GPRS mobile communication systems. KASUMI 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 with 7 bit, the other with 9 bit input/output.

KATAN and KTANTAN [5] 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 our implementation, we consider KATAN with 64-bit block size.

KLEIN [11] 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 it is designed to avoid potential related-key attacks.

mCrypton [15] is a block cipher designed for resource-constrained devices such as RFID tags and sensors. It has a block length of 64 bits and a variable key length of 64, 96 or 128 bits. Here, we implement the variant with 96-bit key length. mCrypton consists of an AES-like round transformation (12 rounds) and a key schedule. The round transformation operates on a 4×4 nibble (4-bit) 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 [6] 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 [4] 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 focus on $\text{SEA}_{96,8}$, i.e. a version of the cipher with 96-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.

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 implementation of the algorithm in C corresponds to about 20 lines of code, and does not involve a 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, e.g., naming functions similar to their original specifications.
2. The cipher has to be implemented in a low-cost way, minimizing the code size and the use of data memory.
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 the 8-bit microcontroller ATtiny45 from ATMEL's AVR device family. It has a reduced instruction set and, e.g., does not feature a hardware multiplier.
7. The encryption and decryption routines have to be called by a common interface.

The SEA reference code was sent as an example to all designers, together with the common interface (also provided at [2]).

The basic metrics considered for evaluation are code size, RAM size, cycle count in en- and decryption, and energy consumption. From these basic metrics, a combined metric is extracted (see Section 6). For the energy-consumption evaluations, each cipher is programmed and executed on an ATtiny45 mounted on a power-measurement board. A 22 Ohm shunt resistor is inserted between the Vdd pin and the 5 V power supply, in order to measure the current consumed by the controller while encrypting. The common interface generates a trigger at the beginning and end of each encryption. The power traces are measured between those two triggers using an oscilloscope that is equipped with a differential probe. We average one hundred encryption traces for each energy evaluation using randomly generated plaintexts and keys for each encryption. The average energy consumed by an encryption is deduced by integrating the measured current.

Finally note that, as mentioned in the introduction, the 12 ciphers are implemented by 12 different designers, with slightly different interpretations of low-cost optimizations. As a result, some of the guidelines could 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, and PRESENT are implemented as 8-bit tables (because of a better time/memory trade-off).

4 Description of the ATtiny45 Microcontroller

The ATtiny45 is an 8-bit RISC microcontroller from ATMEL's AVR series. It 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 256-byte of 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 features a 256-bytes EEPROM memory for non-volatile data storage.

The instruction set of the ATtiny45 consists of 120 instructions which are typically 16-bit 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 or ATxmega devices, the ATtiny45 has a reduced instruction set (e.g., no multiply instruction), smaller memories (Flash, RAM, EEPROM), no in-system debug capabilities, and less peripherals. On the bright side, the ATtiny45 consumes less power and is cheaper in price.

5 Implementation Details

AES. The code is written following the specification for 128-bit key/block size and operates on a state matrix of 16 bytes. In order to improve performance, the state is stored in 16 registers, while the key is stored in RAM. In addition, five temporary registers are used to implement the MixColumn step. The S-box and the round constants are implemented as look-up tables. The multiplication operation needed for MixColumn is computed using shift and XOR instructions.

DESXL. In order to keep code size small, a function which can compute all permutations and expansions depending on the calling parameters is used. This function is also capable of generating 6-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 a straightforward application of the specification. Besides the memory requirements for plain-/ciphertext and the keys k, k_1, k_2, additional 16 bytes of RAM are required for the round key and the state. The S-box and all permutation and expansion tables are stored in Flash memory and are processed directly from there.

HIGHT. First, the intermediate states are stored in RAM at each round and two bytes of the plaintext and one byte of the 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, thus minimizing temporary storage and gaining cycles. Eight subkey bytes are generated once every two rounds and are stored in RAM. Finally, except for the additions mod 2^8 that are replaced by subtractions mod 2^8 and some other minor changes, the same functions as in encryption are used in decryption.

IDEA. This cipher is implemented including a precomputed key schedule performed by separate functions for encryption and decryption, prior to the actual cipher operation. During cipher execution the precomputed key (104 bytes) is then read byte by byte from RAM. The plaintext/ciphertext and the internal state are kept completely in 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 is therefore implemented 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 is written following the functions described in the cipher specifications. During the execution, the 16-byte key as well as the 8-byte running state remain stored in RAM. This allows using only 12 registers and 24 bytes of RAM. Some rearrangements are done to skip unnecessary moves between registers. The 9-bit S-box is implemented as 8-bit table, with the MSBs concatenated in a second 8-bit table. The 7-bit S-box is implemented as 8-bit table, leaving the

MSBs unused in this table. 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 is 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 is 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 are 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 trade-off between code size and processing speed (a similar choice is made for mCrypton and PRESENT, see the next paragraphs). In order 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 registers to the key scheduling, 9 registers are used for temporary storage and 2 registers for the round counter.

mCrypton. The reference code directly follows the cipher specification. The implementation aims for a limited code size. Therefore, as much code as possible is reused for decryption and encryption. In addition, up to 20 registers are used 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 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

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

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 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 is written directly following 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 is implemented using its bitsliced 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 is optimized to limit the RAM usage and code size. As far as RAM is concerned, we only use 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 is 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, make these operations slightly more complex. In particular, 5-position shifts are optimized by replacing them by a 3-position shift in the opposite direction and recovering boundary carries. Nonetheless, TEA proves to be very easy to implement, resulting in a compact code of 648 bytes.

6 Performance Evaluation

We consider 6 different metrics: code size (in bytes), RAM use (in bytes), cycle count in encryption and decryption, energy consumption (in μJ) 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 1, 2, 3, 4, 5, 6 (see appendix). We detail a few meaningful observations below.

First, as our primary goal is to consider compact implementations, we compare our code sizes with the ones listed in [9]. As illustrated in Figure 1, we reduce the memory footprint for most investigated ciphers, with specially strong improvements for DESXL, HIGHT and SEA. The code sizes among our new implementations can also be compared using this figure. 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 when using 8-bit microcontrollers). As expected, standard ciphers such as AES and KASUMI are more expensive, but only up to a limited extent since both can be implemented using less than 2000 bytes of ROM.

The RAM usage in Figure 2 first exhibits the large needs of IDEA regarding this metric (232 words) that is 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 AES, this is in contrast to the "Furious" implementation [18] that uses 192 bytes of RAM and explains our slightly reduced performance for this cipher.

The cycle count in Figure 3 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 leads 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 AES. Cycle count for decryption (Figure 4) shows similar results, with some noticeable changes. Most visibly, IDEA decryption is much less efficient than its encryption. AES also shows an non-negligible overhead when decrypting. In contrast, a number of ciphers behave identical in encryption and decryption, e.g., SEA where the two routines perform almost identical.

As expected, the energy consumption of all the implemented ciphers (Figure 5) is strongly correlated with the cycle count, confirming the experimental results in [8]. However, slight code dependencies can be noticed. This raises an interesting question whether (and to what extend) different coding styles can further impact the energy consumption.

Lastly, the combined metric in Figure 6 first shows the excellent size vs. performance trade-off offered by AES. Among the low-cost ciphers, NOEKEON and TEA exhibit excellent figures as well, most likely 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 trade-off 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 on 8-bit devices as well.

KATAN falls a bit behind, mainly because of its very large cycle count. Only DESXL appears not to be suitable in 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 microcontroller. In total, 12 different ciphers have been implemented with compactness as main optimization criteria. Their source code is available on a web page, under an open-source license. Our results improve most prior work obtained for similar devices. They highlight the different trade-offs 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.

References

1. 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.
2. Anonymous. Implementations of Low-Cost Block Ciphers in Atmel AVR Devices, 2011. Link removed for submission to be anonymous.
3. ATMEL. AVR 8-bit Microcontrollers, http://www.atmel.com/products/avr/.
4. 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.
5. 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.
6. J. Daemen, M. Peeters, G. V. Assche, and V. Rijmen. Nessie Proposal: NOEKEON, 2000. Available online at `http://gro.noekeon.org/Noekeon-spec.pdf`.
7. J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
8. 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.
9. 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.
10. 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 [16], pages 264–278.
11. Z. Gong, S. Nikova, and Y.-W. Law. KLEIN: A New Family of Lightweight Block Ciphers. to appear in the proceedings of RFIDsec 2011.

12. 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.
13. X. Lai and J. L. Massey. A Proposal for a New Block Encryption Standard. In *EUROCRYPT*, pages 389–404, 1990.
14. 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.
15. 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.
16. 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.
17. M. Matsui. New Block Encryption Algorithm MISTY. In E. Biham, editor, *FSE*, volume 1267 of *LNCS*, pages 54–68. Springer, 1997.
18. B. Poettering. RijndaelFurious AES-128 Implementation for AVR Devices, 2007. Available online at `http://point-at-infinity.org/avraes/`.
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 [16], 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.

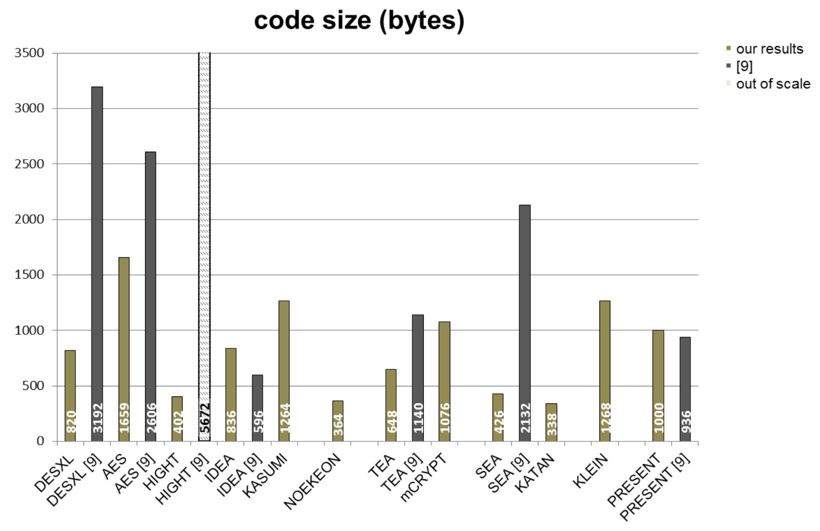

Fig. 1: Code size: comparison with previous work [9].

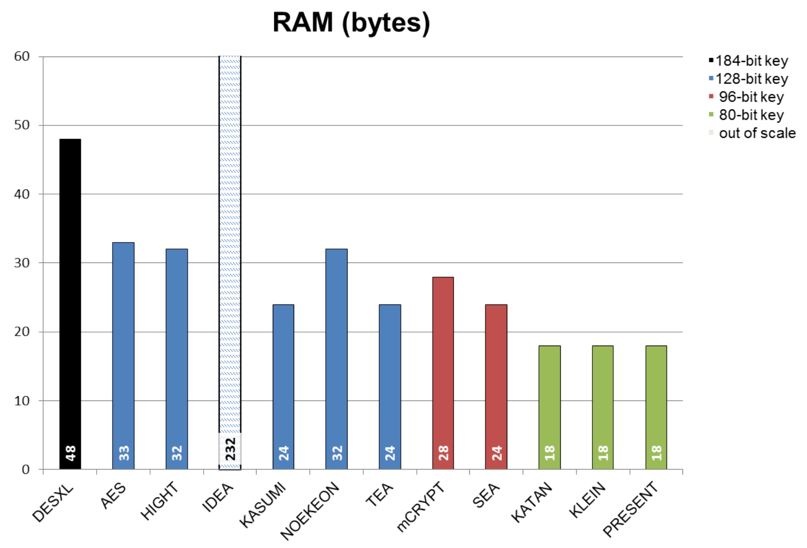

Fig. 2: Performance evaluation: RAM use.

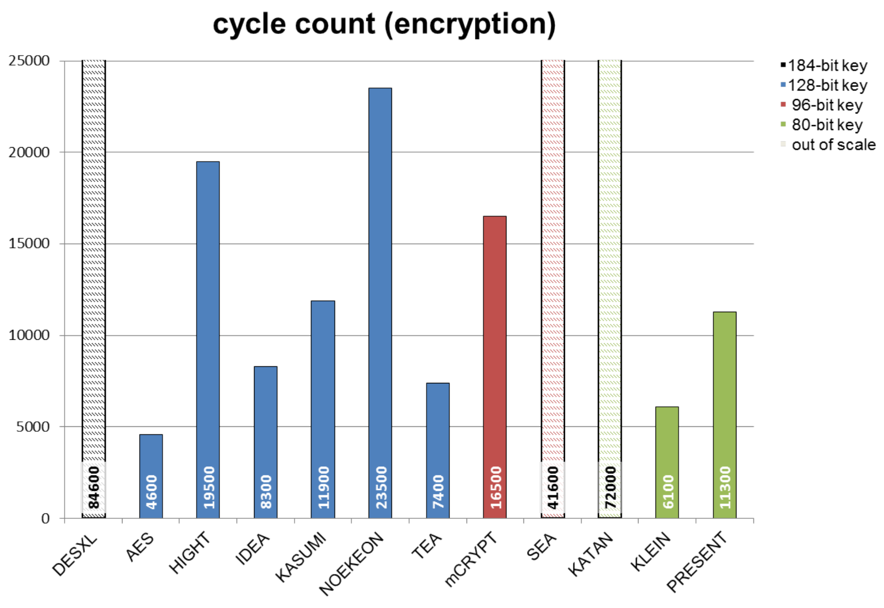

Fig. 3: Performance evaluation: cycle count (encryption).

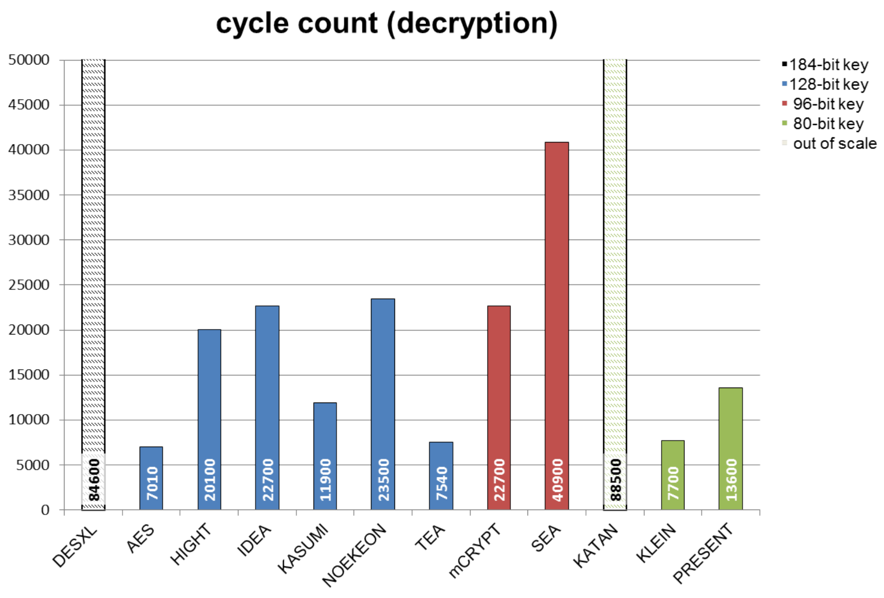

Fig. 4: Performance evaluation: cycle count (decryption).

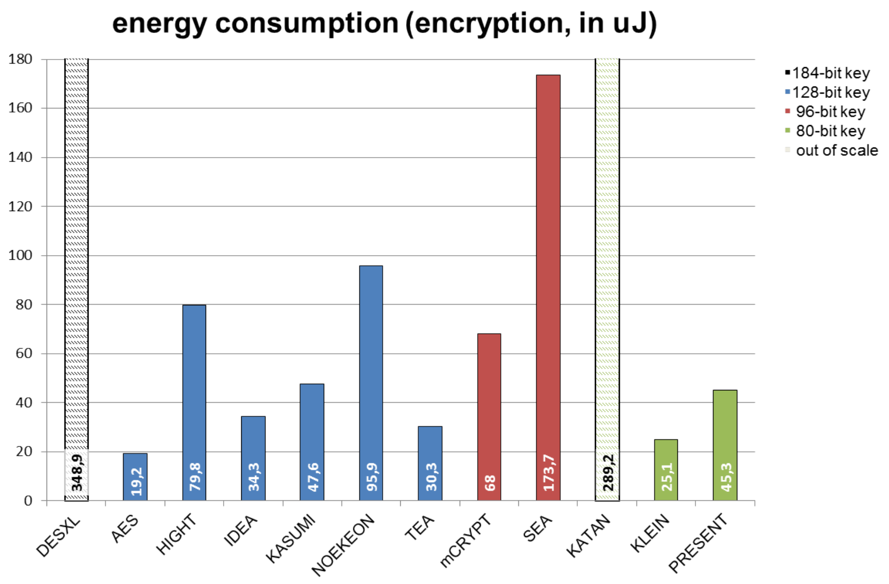

Fig. 5: Performance evaluation: energy consumption.

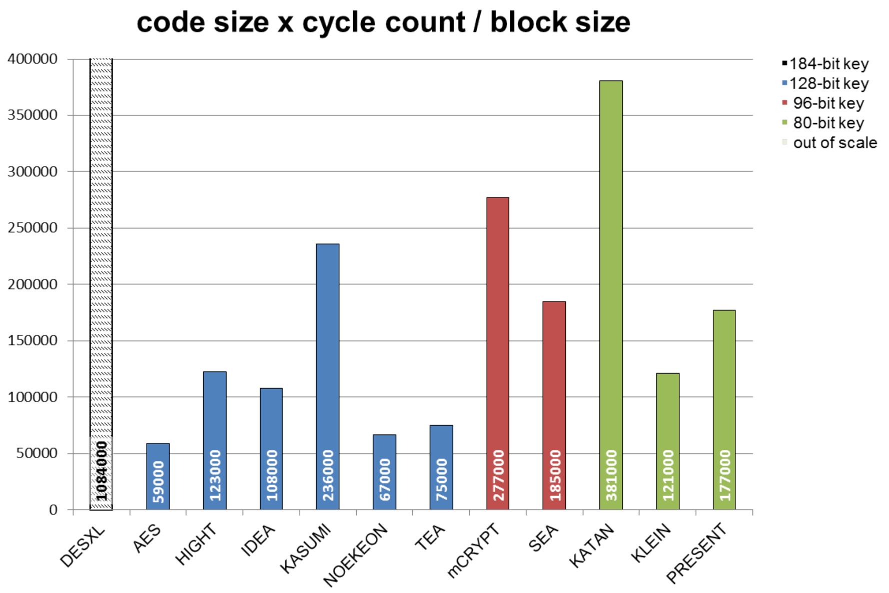

Fig. 6: Performance evaluation: combined metric.