PRESENT Cipher for tinyAVR Architectures

Abstract

This paper presents high-speed and low-size assembly implementations of the 80-bit version of the PRESENT cipher for the AVRtiny family of microcontrollers. We report new speed and size records for our implementation, with the speed-optimized version achieving a full encryption in 8721 clock cycles and the size-optimized version compressing the cipher down to 424 bytes. Along with these two implementation extrema, we offer insight into techniques and representations that utilize the time-memory tradeoff and provide intermediate solutions.

Keywords: PRESENT, AVR tiny, software implementation.

1 Introduction

Modern computer science is constantly witnessing the extensive and large scale deployment of tiny computing devices. Information processing and wireless communication is being thoroughly integrated into everyday objects and activities, developing a large distributed mobile infrastructure and ushering into the era of ubiquitous computing. RFID tags attached on products, cardiac pacemakers, fire-detecting sensor nodes need to operate securely under particularly restricted conditions, namely low battery life, small processing power and bandwidth-demanding ad-hoc network protocols. To achieve sustainable security in this new landscape, researchers have developed new cryptographic primitives and techniques, namely lightweight cryptographic ciphers, such as PRESENT [4], Klein [**?**], LED [**?**] and others. The majority of these ciphers was designed with hardware performance in mind, thus most software implementations are still not efficient. For instance, the AVR-Crypto-Lib [**?**] uses C language implementation resulting in 100k clock cycles for a single encryption with PRESENT. AVR microcontrollers are encountered very often regarding the Internet of Things and ubiquitous computing, thus they compose of an interesting platform to enable and optimize lightweight ciphers. Our contribution consists of a speed-optimized (achieving a full encryption in 8721 clock cycles) and a size-optimized (compressing the cipher down to 424 bytes) implementation of the PRESENT cipher, with the aid of efficient S-Box representations, merged substitution-permutation layers and interleaving of instructions. We aim to provide a range of possible solutions for usage within the AVRtiny architecture.

PRESENT cipher. PRESENT [4] is an ultra-lightweight, 64-bit block symmetric cipher, using 80-bit or 128-bit keys based on a substitution/permutation network and it is named in reference to Serpent [2] due to its similar constructs. As of 2012, PRESENT (among with other ciphers) was adopted by ISO as a standard for a lightweight block cipher (ISO/IEC 29192-2:2012 [12]). The full algorithm has so far been resistant to attempts at cryptanalysis, although the most successful attack has shown that up to 15 rounds can be feasibly broken with $2^{35.6}$ plaintext-ciphertext pairs in 2^{20} operations [1, 6, 14].

PRESENT uses exclusive-or as round key operation, a 4-bit substitution layer, a 4-bit period bit position permutation network in 31 rounds and a final round key operation. Key scheduling is a combination of bit rotation, S-Box application and exclusive-or with the round counter. Constructs found in PRESENT are also encountered in SPONGENT [3], in hash function constructs based on block ciphers proposed by Hirose [5, 9, 10] (H-PRESENT) and in the derivative SMALLPRESENT [13], so the optimizations presented here can also be of interest with respect to the implementations of these ciphers. In our approach, we have implemented PRESENT for the recommended 80-bit key size in AVR assembly in two versions, optimized for maximal speed and minimal size.

PRESENT algorithm. This key register is supplied with the 80-bit cipher key and in every encryption round the first 64 bits of the 80-bit key register form the round key. To encrypt a single 64-bit block, during each encryption round, PRESENT applies a XOR with the current round key followed by a substitution and a permutation layer. The substitution layer applies nibble-wise (4-bit) S-Boxes to the state, while the permutation layer re-arranges the bits in the state following a 4-bit period. Key scheduling is done by rotating the key register 61 bit positions to the left, applying the S-Box to the top nibble of the key register and XORing bits 15 through 19 with the round counter. There is a total of 31 rounds in 80-bit PRESENT and finally the round key is applied one last time (Figure 1.).

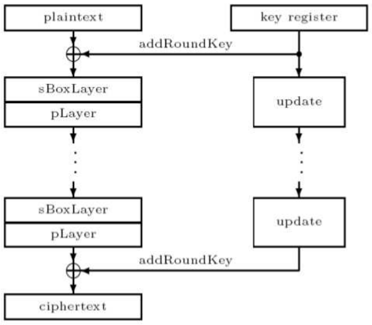

Figure 1: Schematic of the PRESENT cipher. It consists of 31 rounds, including substitution, permutation and key update.

Paper organization. The organization of the paper follows. Section 2 provides an introduction to the AVR architecture, including the capabilities and limitations of AVRtiny microcontrollers. Section 3 focuses on high-speed implementation of the PRESENT cipher, regarding efficient S-Box representations and lookup tables. Section 4 describes various techniques that produce a very compact version of the cipher. Section 5 provides the final benchmarking results and conclusions.

2 The 8-bit family of tinyAVR Microcontrollers

Atmel offers a wide range of 8-bit microcontrollers, including high performance devices (ATxmega), mid-range devices (ATmega) and low-end devices with limited memory, storage and processing power (ATtiny). Common applications of AVR microcontrollers include smart cards, motor control systems, medical applications et cetera.

Our focal point is the resource-constrained ATtiny architecture, which typically features less than 1 kilobyte of static RAM (SRAM) and flash storage ranging from 1 to 8 kilobytes. The architecture uses 32 general-purpose registers, *R0-R31*. Several registers have special characteristics, namely registers *R0* through *R15* can only be accessed by instructions that provide an immediate value as an operand, *e.g.* you cannot perform memory access from these registers. In addition, registers *R27:R26* (denoted as X) and registers *R29:R28* (denoted as$Y$) can be used for accessing the SRAM, while registers *R31:R30* (denoted as Z) can access the flash storage. At all times, special memory registers can be utilized as general-purpose registers.

The ATtiny instruction set consists of the basic 90 instructions, found in all AVR architectures. However, due to the limited size of its core, it does not support the extended instruction set which includes multiplication, in contrast with the ATmega architecture.

ATtiny configuration. We perform all simulations on ATtiny45, which has a maximum clock frequency of 20 MHz, 256 bytes of SRAM and 4 KB of flash storage.
Radix-2^8 representation. The PRESENT ciphers requires representing integers of size 64 and 80 bits. Thus, we split the number into 8-bit components, using radix-2^8 and an m-bit integer is represented as $n = \lceil m/8 \rceil$ bytes $(x_0, x_1, \ldots, x_{n-1})$ such that $x = \sum_{i=0}^{n} x_i 2^{8*i}$.

3 High-speed implementation

3.1 PRESENT S-Box & Player implementation

In this section we examine the S-Box of the PRESENT cipher from the speed perspective, using lookup tables, and we offer several variations, utilizing the time-space tradeoff. We analyze the *original S-Box*, identify its performance issues (section 3.1.1) and suggest two possible lookup table representations (with 8 and 16 bytes respectively). Subsequently, we expand it to the faster, yet space-demanding (256 bytes) *Squared S-Box* (section 3.1.2). In the last section, we examine the *combination of the S-Box and the permutation layer*, resulting in a very large lookup table (1024 bytes) that substantially boosts performance (section 3.1.3).

3.1.1 Original S-Box

The original S-Box, presented by Bogdanov et al. [4] consists of 16 different substitutions, each with a 4-bit input and a 4-bit output, as shown in Table 1.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 1: The original S-Box of the PRESENT cipher.

Representation. In the case that we aim for particularly low size footprint, it is possible to use either a *packed* or an *unpacked* representation of the original S-Box. The *unpacked* version stores the lookup table in 16 bytes, where every 4-bit input to 4-bit output substitution is stored using 8 bits of space (*i.e.* there exists redundancy in the representation). The *packed* version, stores *two* 4-bit input to 4-bit output substitutions using 8 bits, *i.e.* without any redundancy, resulting in an 8 byte lookup table. (See Table 2)

x	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
S[x]	0C	05	06	0B	09	00	0A	0D	03	0E	0F	08	04	07	01	02

Table 2: The unpacked representation of the original S-Box, using 16 bytes. Each table column represents a single substitution.

x	01	23	45	67	89	AB	CD	EF
S[x]	C5	6B	90	AD	3E	F8	47	12

Table 3: The packed representation of the original S-Box, using 8 bytes. Each table column represents two substitutions. This would give a size optimization of 8 bytes to begin with, but considerations for unpacking code apply. (See Section 4.2.)

Performance. The core performance issue regarding the 4-bit S-Box is the penalty in accessing it, if stored in a lookup table. The AVR architecture is designed to enable fast access for 8-bits at a time. Thus, a lookup table with of the original S-Box is rather small (16 bytes for an unpacked version, 8 bytes for a packed one), but it is also rather inefficient (speed-wise) due to the overhead operations that need to take place before and after each table lookup. The packed S-Box (Table 3) is the least speed-efficient variant, since after the lookup we always have to extract the upper or lower half. This issue is not encountered in the unpacked version (Table 2), which makes better use of the AVR 8-bit architecture. However, performance is not optimal due to the fact that we only substitute 4 bits at a time, while we use 8-bit operations, *i.e.* the redundant representation results in more memory accesses than needed.

3.1.2 Squared S-Box

A solution to the afore mentioned performance problems is to construct a new lookup table that:*(a)* is custom made for the 8-bit AVR architecture, similarly to the unpacked S-Box and *(b)* uses a non-redundant representation similarly to the packed S-Box.
Representation. In Table 4, we demonstrate a *Squared S-Box*, which uses an 8-bit input and produces an 8-bit output. Within an 8-bit space, we can contain two 4-bit substitution values and the number of possible substitution values is 16, thus the total size of the lookup table is $16 * 16 = 256$ bytes.

As a result, there is no need for overhead computation and the substitution consists of a single lookup. This approach has also been followed before by researchers [7]

Performance. The Squared S-Box described, is an efficient and viable solution with respect to the cipher's substitution layer. It is custom made for the 8-bit AVR architecture and allows us to perform byte substitutions with a single flash memory lookup. Furthermore, it is relatively size-efficient, consisting of 256 bytes, thus, it can be transfered to ATtiny45 SRAM from flash memory during the initialization process of the algorithm. The memory transfer is viable, since ATtiny45 possesses 256 bytes of SRAM and it will reduce the lookup cost by 33%, *i.e.* to perform a lookup, the software will use the `ld` instruction (load from SRAM - 2 clock cycles), instead of the `lpm` instruction (load from flash memory - 3 clock cycles) and thus, each lookup will cost 1 cycle less. Given the fact that the block size of the PRESENT cipher is 64 bits, it requires $64/8 = 8$ lookups per round. The cipher consists of 31 rounds and an implementation using SRAM will result in $8 * 31 = 248$ clock cycles reduction for a full encryption. The only processing penalty consists of a 256 byte transfer from flash storage to SRAM which occurs only once, preferably in the beginning or setup phase.

x	00	01	02	03	...	0C	0D	0E	0F
S[x]	CC	C5	C6	CB	...	C4	C7	C1	C2
x	10	11	12	13	...	1C	1D	1E	1F
S[x]	5C	55	56	5B	...	54	57	51	52
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮	⋮	⋮
x	F0	F1	F2	F3	...	FC	FD	FE	FF
S[x]	2C	25	26	2B	...	24	27	21	22

Table 4: The 256-byte Squared S-Box. It substitutes one byte at a time, without any overhead or redundancy.

3.1.3 Merged SP lookup table

Although the Squared S-Box lookup table solution is viable for the PRESENT cipher, we need to stress the fact that the PRESENT cipher possesses a complex permutation layer, requiring a large number of shift and rotate operations. The AVR architecture is not capable of performing fast shifts and rotations (compared for instance with the ARM11 processors which can do shifts and rotations for free), *i.e.* an n-bit shift requires n clock cycles, and thusly we have to look for alternatives. In this direction, work by Hutter &

Schwabe [11] on ATmega[1] suggested the usage of multiplications instead of rotations/shifts, however this is not viable on ATtiny due to the fact that they do not possess native multiplication instructions.

Representation. The fastest approach that we identified for the permutation layer is the idea developed by Zheng Gong and Bo Zhu [16, 8]. Specifically, they exploited the internal structure of the permutation layer, *i.e.* the fact that every output of a 4-bit S-Box will contribute one bit to the cipher (the underlying pattern for the permutation is the following: $for\ k$: *old position*, l : *new position*, $l = f(k) = 16(kmod4) + (kdiv4)$). Thus, the first 2 bits of the output are derived from the first two 4-bit S-Boxes, i.e. from the first byte of the previous state.

Using these observations, they crafted four 256-byte lookup tables (1024 bytes in total) that *merge* the S-Box and the permutation layer and as a result, the whole SP network is performed via table lookups.

Performance. The 1024 byte lookup tables eliminate the need for an independent permutation layer, providing us with the fastest available solution. On the downside, we have to perform one lookup for every two bits, resulting in 32 lookups for a 64-bit state (compared to the Squared S-Box that required only 8 lookups for a 64-bit state). Moreover, we need 1024 bytes to store the tables, thus it is not possible to transfer them to the SRAM. The 33% speedup obtained in section 3.1.2 is only possible in an AVR microcontroller with at least 1024 bytes of internal[2] SRAM, for instance ATtiny1634.

Time-memory tradeoff. We conclude the discussion on the substitution/permutation techniques by presenting all the proposed solutions (original, squared or merged S-Box) in the form of a spectrum diagram. Depending on the requirements, resources or limitations induced, one may choose the most suitable representation, with the aid of the spectrum diagram and the following table.

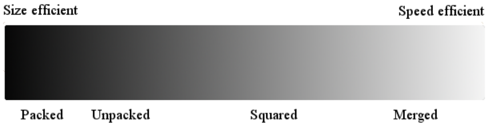

Figure 2: The available S-Box representations, shown on the size-speed spectrum.

[1] Benchmarking was performed on ATmega2560.

[2] Additional external SRAM is not an option, since it is at least as slow as flash memory.

Representation	Flash (bytes)	SRAM (bytes)	Comments
Packed S-Box	8	0-8	Compact, slow
Unpacked S-Box	16	0-16	Compact, slow
Squared S-Box	256	0-256	Medium speed and size
Merged SP	1024	0-1024	High speed and size

Table 5: Comparison of S-Box representations. Note that all representations are viable on ATtiny45 when implemented with flash storage. In the case that the tables are transfered to SRAM for faster lookups, only packed, unpacked and squared are viable on ATtiny45 due its SRAM size (256 bytes). Merged SP on SRAM is viable on ATtiny1634 (featuring 1KB of SRAM).

3.1.4 Memory optimizations for lookup tables

All the afore-mentioned solutions with respect to substitution and/or permutation rely heavily on lookup tables. In order to decrease the computational penalty of the table lookups, we performed several code-level optimizations. Below, we demonstrate the code required to perform a single table lookup from flash memory.

```
mov ZH , high_part_of_address;
mov ZL , low_part_of_address;
lpm register , Z ;
```

The lookup operations consists of two `mov` and one `lpm` instruction. Memory is adresses by using 16 bits, so the first `mov` loads the high part, while the second `mov` loads the table index that will be accessed.

Table alignment. We aim to keep the changes required in the high part (`ZH`) to a minimum. Thus, we align the four 256-byte tables required for the merged SP approach in section 3.1.3 such that they can be accessed by using only the low part (`ZL`) register as an index and keeping `ZH` unchanged. Elaborating, the four lookup tables start from `0x0600, 0x0700, 0x0800, 0x0900` respectively and thus, the 8 high bits of the address part (`0x06, 0x07, 0x08, 0x09`) remain the same while the 8 low bits are sufficient to act as the table index, ranging from 0 to 255 (`0x00` to `0xFF`).

Memory access grouping. Performing two lookup operations in two different tables requires a total of 10 clock cycles (2 `mov` to change the high part of the operation, 2 `mov` to change the index and 2 `lpm` to perform the actual lookup). However, performing two lookup operations on the *same table* requires a total of 8 clock cycles (2 `mov` for the index, 2 `lpm` for the lookup), since the high part of the memory address remains the same. Thus, we we try to perform the maximum amount of grouped table lookups, given the limited number of registers, since within each group, `ZH` remains the same. For instance the following sequence of operations, *lookupTable1(i), lookupTable2(k), lookupTable1(j), lookupTable2(m)* will transform to *lookupTable1(i), lookupTable1(j), lookupTable2(k), lookupTable2(m)* in order to group memory access.

3.2 Key update implementation

This section focuses on implementing the key scheduling/update process of the PRESENT cipher efficiently. The key update function of the PRESENT cipher consists of three operations, namely, key rotation, key substitution and key XOR the round counter. We present the optimizations performed in the following subsections.

3.2.1 Key rotation

The algorithm specifies that the key must be rotated by 61 bits to the left. Given the fact that rotations/shifts are computationally expensive in the AVR architecture, we transform 61 left rotations to 19 right rotations, which can be further reduced to 16 right rotations and 3 right rotations. The 16 right rotations can be easily performed by using the `mov` instructions on register level, i.e. rotate all the bits inside a register by moving the contents to the previous register used in our representation, an approach which is preferable to single rotations via the bit-level instructions. Only the 3 remaining rotations are carried out with the logical instructions for right rotation and shifting (`ror` and `shr`).

3.2.2 Key substitution

The highest 4 bits of the 80-bit key used by the PRESENT cipher, must go be substitute via the S-Box. To avoid 4-bit memory access or redundancy (section 3.1.1), we construct a special-purpose Squared S-Box that substitutes the 4 high bits of the 8-bit input, while the low 4 bits remain unchanged. The resulting Table 6 applies a substitution operation on the upper nibble which takes only a single lookup operation. In the case that we have space constraints, it is possible to replace the Squared S-Box with the original, unpacked one; the key substitution occurs only once per round, so the performance loss incurred by the unpacked S-Box is relatively small.

x	00	01	02	03	...	0C	0D	0E	0F
S-Box[x]	C0	C1	C2	C3	...	CC	CD	CE	CF
x	10	11	12	13	...	1C	1D	1E	1F
S-Box[x]	50	51	56	5B	...	5C	5D	5E	5F
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮	⋮	⋮
x	F0	F1	F2	F3	...	FC	FD	FE	FF
S-Box[x]	20	21	22	23	...	2C	2D	2E	2F

Table 6: Special-purpose Squared S-Box for key substitution, operating only on the high 4 bits of the input.

3.2.3 Key exclusive OR operation

The algorithm specifies that the key bits 15, 16, 17, 18, 19 must be XORed with the round counter. The issue is that -under the current representation- bits 0. . .7 will be stored in register0, bits 8. . .15 will be stored in register1 and bits 16. . .23 will be stored in register2. As a result parts of the round counter need to be XORed with different parts of two separate registers, namely the counter needs to be XORed with both register1 and register2. Similarly to [7], we perform the XOR operation before the key rotation, thus the bits that are operated on are bits 34,35,36,37,38 which span a single register (using the previous representation, they are located in register4). This restructuring, *i.e.* performing the XOR operation before the key rotation does not affect the outcome or security of the algorithm.

4 Size optimizations

Here we will list some of the size improvements we were able to apply to the PRESENT algorithm. While these modifications make the algorithm operate more slowly, the reduction in size would allow the cipher to be included in microcontrollers with smaller available code area. Our version requires 204 AVR instructions (408 bytes of code) for both the encryption and decryption routines, plus two times 8 bytes for packed tables of S-box values at memory addresses `0x100` and `0x200`. We believe this should be sufficiently small for the code to be included in an AVR machine with 1K of available Flash storage, while still allowing over half of the available area to be devoted to application-specific code.

Unfortunately, every opcode in the AVR instruction set is 16-bits wide, so there is nothing to be gained from exchanging any instructions for equivalent smaller ones (such as for example *add Rd, 1* being more concisely expressed as *inc Rd* on x86 machines). Furthermore the AVR employs a

Harvard architecture, where there is a strict separation between data and code memory; this prevents us from dynamically computing new opcodes in memory to be executed later. Finally we note there are no 'bulk' instructions which operate on several registers at once.

However, we do have access to some instructions that are specific to the AVR architecture and are uniquely suited to making parts of the code more condensed by virtue of their expressiveness, such as the *swap* and *cbr* instructions.

4.1 Serialization

The greatest size optimization we have implemented is serialization of the algorithm, keeping part of the state in SRAM while we operate on fewer registers. This reduces the instruction count on all parts of the round update procedure except for key scheduling.

We have chosen to keep 4 bytes of state in registers at a time, as we believe it allows us to apply the permutation layer of the algorithm in software with the most size-efficiency. Using even fewer state bytes in registers might allow for greater size reductions in other areas of the code, but applying the permutation layer may turn out to be more troublesome in that case.

4.2 S-box packing

The PRESENT S-boxes work on 4-bit nibbles, but defining a table of nibbles in the code at first seemed less size-efficient than packing the nibbles into bytes to be unpacked. This would save 8 bytes per S-box table to start with, but we need 5 to 8 instructions in stead of a **ld** (load) depending on whether or not we care about timing attacks to unpack the nibbles which diminishes the size benefit to 2 bytes of code. See Figure 3 and Table 3.

4.3 Permutation layer

The code to apply permutations to the state in software borrows heavily from the AVR implementation of PRESENT drafted in Leuven by Eisenbarth et al [7]. Since the permutation follows a 4-bit period in the input it seems most efficient to use 4 bytes of I/O when rotating bits off of registers into corresponding new positions.

In the Leuven implementation one bit is rolled off from 4 state registers into one output register and this block is done twice, completing one output byte. It has made ingenious use of the side-effect that both the input and the

```
unpack_sBox:
        asr ZL                    ; halve input, take carry
        lpm SBOX_OUTPUT, Z        ; get S-box output
        brcs odd_unpack           ; branch depending on carry
even_unpack:
        swap SBOX_OUTPUT          ; swap nibbles in S-box output
        rjmp unpack
odd_unpack:
        nop                       ; guard against timing attacks
        nop
unpack:
        cbr SBOX_OUTPUT, 0xf0 ; clear high nibble in S-box output
```

Figure 3: Unpacking bytes into nibbles in a constant cycle count takes us 8 instructions whereas unpacking unpacked nibbles takes us only 1. The net gain is only 2 bytes of code.

output are being rotated, allowing the operation to be reversed depending on the offset at which the procedure is called.

4.4 Branching on SREG flags

A subtle but real size optimization we implemented focused on the code used to drive repeated operations. The construct in Figure 4 allows the implementer to let a block of code be executed twice, while allowing them to take control of the machine before, after and in between these code blocks. In our implementation this has been applied to the substitution and permutation layer procedures to save a few instructions. This does not affect the cycle count relative to the state or input. This construct also allows the *unpack_sBox* code we just defined to be inlined into our S-box procedure, saving 2 more instructions.

We have used this construct in the procedures for both steps of applying the SP-network while retaining the property that these procedures can be called from two different offsets.

The S-box substitution procedure for example replaces only the high nibble when called from one offset, and the whole byte when called from another. The code to swap a byte simply re-uses a nibble-wise operation while swapping the nibbles before and after to complete an entire byte.

In the permutation procedure the operations on 4 input bits is done twice, to complete one output byte.

```
setup_redo_block:
        clt                         ; clear T flag
        rjmp  redo_block            ; do the second part
block:
        set                         ; set T flag
        ; fall through
redo_block:
        ; instructions here happen twice when called from block

        brts  setup_redo_block  ; redo this block? (if T flag set)
        ret
```

Figure 4: A construct that uses the state register to re-do a block twice with code executing before, after and in between, allowing more code to be inlined.

4.5 Keying and key scheduling

Because we have serialized part of the algorithm our key register needs to be rotated when changing the context between steps that apply to the higher and lower 4 bytes of the cipher's state. Because we required a procedure to rotate the key register for key scheduling anyway, that code can be reused for this purpose, and we can still apply the exclusive-or to part of the key register in the ideal position (i.e. where the bytes of the key register line up with the round counter register).

The inverse key scheduling procedure is only needed in the decryption routine, so we were able to inline it into the decryption round and combine the operations for key scheduling and rotating the register to the appropriate position for the next round.

4.6 Reducing code size for specific applications

While the attained size of the implementation of PRESENT should suffice for use in real-world applications, some of the input and output procedures as well as the S-box unpacking code can be omitted when only encryption or encryption is required in the application. This is highlighted in the source code using comments.

5 Conclusion

We've compared our results to the existing AVR implementation by Eisenbarth et al [7] and the GNU AVR-Crypto-Lib [15] (which was written in standard C and is only provided as a baseline reference). We are pleased

	GNU reference	Leuven reference	Speed optimized	Size optimized
Size (bytes)	1514	936	1794	424
Encryption (cycles)	105796	10723	8721	93887
Decryption (cycles)	151624	11239	-	105605
RAM (bytes)	256	0	18	18

Table 7: Speed and size comparisons to existing implementations of PRESENT for the AVR architecture.

to announce we were able to halve the code size and gain about 13% speed increase. (See Table 7.) We note that having access to a larger SRAM could allow the lookup tables for the speed-optimized version to incur a lower overhead, and reduce an estimated 10% more cycles per encryption.

References

[1] Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel. Biclique cryptanalysis of the present and led lightweight ciphers. Technical report, Cryptology ePrint Archive, Report 2012/591, 2012.

[2] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the advanced encryption standard. *NIST AES Proposal*, 1998.

[3] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. Spongent: A lightweight hash function. *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 312–325, 2011.

[4] Andrey Bogdanov, Lars Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 450–466, 2007.

[5] Andrey Bogdanov, Gregor Leander, Christof Paar, Axel Poschmann, Matt JB Robshaw, and Yannick Seurin. Hash functions and rfid tags: Mind the gap. In *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 283–299. Springer, 2008.

[6] Baudoin Collard and F-X Standaert. A statistical saturation attack against the block cipher present. In *Topics in Cryptology–CT-RSA 2009*, pages 195–210. Springer, 2009.

[7] Thomas Eisenbarth, Zheng Gong, Tim Güneysu, Stefan Heyse, Sebastiaan Indesteege, Stéphanie Kerckhof, François Koeune, Tomislav Nad, Thomas Plos, Francesco Regazzoni, et al. Compact implementation and performance evaluation of block ciphers in attiny devices. *Progress in Cryptology-AFRICACRYPT 2012*, pages 172–187, 2012.

[8] Zheng Gong, Pieter Hartel, Svetla Nikova, and Bo Zhu. Towards secure and practical macs for body sensor networks. *Progress in Cryptology-INDOCRYPT 2009*, pages 182–198, 2009.

[9] Shoichi Hirose. Provably secure double-block-length hash functions in a black-box model. In *Information Security and Cryptology–ICISC 2004*, pages 330–342. Springer, 2005.

[10] Shoichi Hirose. Some plausible constructions of double-block-length hash functions. In *Fast Software Encryption*, pages 210–225. Springer, 2006.

[11] Michael Hutter and Peter Schwabe. NaCl on 8-bit AVR microcontrollers.

[12] Information technology - Security techniques - Lightweight cryptography - Part 2: Block ciphers, 2011.

[13] Gregor Leander. Small scale variants of the block cipher present. *IACR ePrint Report*, 143, 2010.

[14] Jorge Nakahara Jr, Pouyan Sepehrdad, Bingsheng Zhang, and Meiqin Wang. Linear (hull) and algebraic cryptanalysis of the block cipher present. In *Cryptology and Network Security*, pages 58–75. Springer, 2009.

[15] The GNU project. Avr-crypto-lib. `http://avrcryptolib.das-labor.org/trac`, 2013. [Online; accessed 6-April-2013].

[16] Bo Zhu Zheng Gong. Software implementation of block cipher present for 8-bit platforms. `http://cis.sjtu.edu.cn/index.php/Software_Implementation_of_Block_Cipher_PRESENT_for_8-Bit_Platforms`, 2013. [Online; accessed 19-February-2013].