Speed and Size-Optimized Implementations of the PRESENT Cipher for Tiny AVR Devices

Konstantinos Papagiannopoulos and Aram Verstegen

Radboud Universiteit Nijmegen

Abstract. This paper presents high-speed and low-size assembly implementations of the 80-bit version of the PRESENT cipher for the (Tiny)AVR family of microcontrollers. We report new speed and size records for our implementations, with the speed-optimized version achieving a full encryption in 8721 clock cycles and the size-optimized version compressing the cipher down to 272 bytes; the previous state of the art for (Tiny)AVR achieved 10723 clock cycles for encryption with a size of 936 bytes. Along with the two implementation extrema (speed & size optimized versions), we offer insight into techniques and representations that show the speed/area tradeoffs and provide intermediate solutions for various configurations.

Key words: PRESENT, AVR, ATtiny, Assembly Software Implementation

1 Introduction

Modern society is constantly witnessing an extensive and large scale deployment of tiny computing devices. Information processing and wireless communication are being thoroughly integrated into everyday objects and activities, developing a large distributed mobile infrastructure and ushering in the era of ubiquitous computing. RFID tags attached to products, cardiac pacemakers, fire-detecting sensor nodes and the like 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 **?**, Klein **?**, LED **?** and others.

The majority of these ciphers was designed with hardware performance in mind, leaving most software implementations relatively inefficient. For instance, the AVR-Crypto-Lib **?** often resorts to C language implementations, resulting in 100.000 clock cycles for a single encryption with PRESENT. AVR microcontrollers are often encountered regarding the Internet of Things and ubiquitous computing, thus they are an interesting platform on which to enable and optimize lightweight ciphers. The University of Louvain has initiated a project to draft efficient assembly implementations of various lightweight ciphers on resource-constrained AVR devices to make fair comparisons about their relative efficiency. Resulting from this project is the state of the art in both speed and size: an implementation **?** which achieves an encryption with PRESENT in 10723 clock cycles in 936 bytes.

Our contribution. This paper describes the details of a speed optimized **?** and a size-optimized **?** implementation of the PRESENT cipher on the ATtiny45, attained with the aid of algorithmic improvements and efficient programming techniques. Our speed-optimized version improves the state of the art **?** by an 18% reduction in clock cycles, while the size-optimized version is 70% smaller.

Algorithmic improvement:

- A merged SP layer, *i.e.* combining the substitution and permutation layer of the cipher in order to construct lookup tables that remove the time-consuming permutations **??**. This optimization constitutes the core of the achieved speed improvement.

Programming improvements:

- Squared S-Box representations, *i.e.* S-Boxes which are custom-made for fast access in the AVR 8-bit architecture, also used by Eisenbarth **?**.
- Compact S-Box representations, *i.e.* minimal footprint S-Boxes that reduce the implementation size.
- Minimal key register rotations, allowing the key update procedure to complete in fewer instructions.
- Memory access optimizations, grouping memory transactions to improve speed.
- Algorithm serialization, by keeping part of the state in SRAM while we operate on fewer registers.
- Indirect register access to let loops drive repeated operations on CPU registers.
- Use of the stack to store intermediate values to avoid using more dedicated registers.
- Code restructuring and efficient procedure callings.

2 Background

2.1 PRESENT Cipher

PRESENT **?** is an ultra-lightweight, 64-bit symmetric block cipher, using 80-bit or 128-bit keys. It is based on a substitution/permutation network and it is named as a reference to Serpent **?** due to its similar constructs. As of 2012, PRESENT (among other ciphers) was adopted by ISO as a standard for a lightweight block cipher (ISO/IEC 29192-2:2012 **?**). The full algorithm has so far been resistant to attempts at cryptanalysis, although the most successful attack has shown that up to 15 of its 31 rounds can be broken with $2^{35.6}$ plaintext-ciphertext pairs in 2^{20} operations **???**.

PRESENT uses exclusive-or as its 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 **?**, in hash function constructs based on block ciphers as proposed by Hirose **???** (H-PRESENT) and in the similar Maya **?** or generalized SMALLPRESENT **?**. Thus 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. Support for 128-bit keys was also added to the size-optimized implementation as the required extra registers became available through optimizations, but we will focus on the implementation of the variant with 80-bit keys.

2.2 PRESENT Algorithm

The cipher's 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 an exclusive-or (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 such rounds and finally the round key is applied one last time (Figure **??**.).

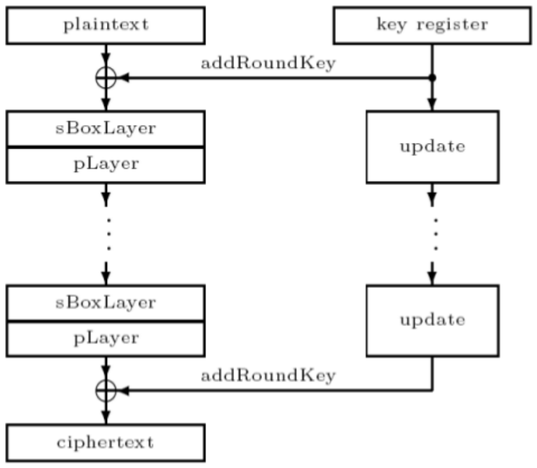

Fig. 1. Schematic of the PRESENT cipher. It consists of 31 rounds, including XOR round key application, nibble-wise substitution, bit position permutation and key update.

2.3 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 focus is on 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 not be accessed by instructions that provide an immediate value as an operand, *i.e.* you can only perform memory access from these registers. In addition, register pairs *R27:R26* (denoted as X), *R29:R28* (denoted as Y) , and *R31:R30* (denoted as Z) can access the SRAM. The Y register can also be used for indirect register addressing, the Z register can also access the flash storage and be used for indirect jumps and calls. Instructions using these 3 pointer register pairs also allow post-increment and pre-decrement of the pointer. At all times, the 6 special registers can be utilized as general-purpose registers.

The ATtiny instruction set consists of the basic 90 single-word 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 the 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 cipher 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 & P-Layer 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 speed-area trade-off. We analyze the *original S-Box*, identify its performance issues 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.* 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.

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

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. If we aim for a particularly small size footprint, it is possible to use either a *packed* or an *unpacked* representation of the original S-Box. The original, *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 of 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.

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

Table 2. 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 **??**.)

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 of the original S-Box is rather small (16 bytes for an unpacked version, 8 bytes for a packed one), but we can assume it is also relatively inefficient speed-wise due to the overhead operations that need to take place before and after each table lookup. Surely the packed S-Box (Table **??**) is the least speed-efficient variant, since after the 4-bit lookup we also have to extract the upper or lower half. This issue is not encountered in the unpacked version, 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.

Squared S-Box A solution to the aforementioned performance problem is to construct a new lookup table that: *a)* is custom made for the 8-bit AVR architecture, like the unpacked S-Box and *b)* uses a non-redundant representation similar to the packed S-Box.
Representation. In Table **??**, 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 $16 \cdot 16 = 256$ 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 Eisenbarth **?**

Performance. The Squared S-Box described is an efficient and viable solution with respect to the cipher's substitution layer. It is cus-

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

tom 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. We consider it could almost be transfered to ATtiny45 SRAM from flash memory during the initialization process of the algorithm, but we would be left without any stack space and therefore it would certainly be a special purpose implementation. That memory transfer *is* viable for applications that do dedicated encryption or decryption (provided we are left with some stack space for the workings of the rest of the algorithm). The instruction to load from flash (`lpm`) takes 3 clock cycles, whereas loading from or saving to SRAM (`ld`) takes only 2. Given the fact that the block size of the PRESENT cipher is 64 bits, it requires $64/8 = 8$ S-Box lookups per round. The cipher consists of 31 rounds so a full encryption requires $8 \cdot 31 = 248$ S-Box applications. If an application would singularly encrypt or decrypt, we could save 248 cycles per encryption after an initial $(3 + 2) \cdot 256 = 1280$ clock cycles. That means that the processing penalty to load a 256 byte table from flash storage to SRAM would not be compensated until over 5 encryptions or decryptions were computed sequentially, so this approach is not feasible for the general case of intermixed encryption and decryption.

Merged SP Lookup Table Although the Squared S-Box lookup table solution is viable for the PRESENT cipher, we need a speed optimization for the most complex part of the PRESENT algorithm: the permutation layer. In contrast to hardware implementations, where it is a trivial rewiring of outputs, most software implementations require a large number of bit rotation and move operations. The AVR architecture is not capable of performing fast shifts and rotations (compared to for instance 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 **?** on ATmega[1] suggested the usage of multiplications instead of rotations/shifts, however this is not viable on ATtiny chips 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 **??**. Due to the adaptation of this novel approach in the AVR architecture, we are able to improve performance over the state of the art **?**. Specifically, Gong & Zhu 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:

$$for\ k : old\,position,\ l : new\,position,\ l = f(k) = 16\cdot(k \bmod 4)+(k \div 4)$$

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 on our test platform. The theorized 33% speedup considered with using SRAM instead of flash storage is only possible in an AVR microcontroller with at least 1024 bytes of internal[2] SRAM, for instance ATtiny1634.

Memory Optimizations for Lookup Tables All the aforementioned 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. In Figure **??** we demonstrate the code required to perform a single table lookup from flash memory.

[1] Benchmarking was performed on ATmega2560.

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

```
mov ZH, high      ; load high part of Z address
mov ZL, low       ; load low part of Z address
lpm register , Z  ; load from Z addresses into register
```

Fig. 2. Flash memory is addressed through 16-bit pointers. We aim to keep changes in the high byte to a minimum.

The lookup operation consists of two `mov` and one `lpm` instruction. Memory is addressed using 16-bit pointers, 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 **??** 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.

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`).

Key Substitution The highest 4 bits of the 80-bit key used by the PRESENT cipher, must be substituted via the S-Box. To avoid 4-bit memory access or redundancy (section **??**), 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 applies a substitution operation on the upper nibble which takes only a single lookup operation. Should we encounter 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.

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 *R0*, bits 8. . .15 will be stored in *R1* and bits 16. . .23 will be stored in *R2*. 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 *R1* and *R2*. Similarly to Eisenbarth **?**, 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 (under the previous representation they are located in *R4*). This restructuring of the algorithm, *i.e.* performing the XOR operation before the key rotation, does not affect the outcome or security of the algorithm.

Latency *vs.* Throughput The implementations presented focus on reducing the cost of a *single* encryption. Thus, it can also be viewed as a *low-latency* implementation of PRESENT. Should we loosen up on the latency requirement, we can achieve increased *throughput*. Future work aims to perform multiple encryptions at a time, by using the *bitslicing* technique **?**, which largely reduces the cost of permutations.

4 Size-Optimized Implementation

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 128 AVR instructions (256 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 almost three quarters of the available area to be devoted to application-specific code.

Unfortunately, every opcode in the AVR instruction set is expressed using one or more 16-bits words, so while using only the single-word instructions of the ATtiny there is no further possibility to exchange 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. We also have all kinds of branching instructions at our disposal that branch based on values in the state register (SREG) which we can explicitly or implicitly modify. We can use the stack to make procedure calls or as temporary storage, and finally have the powerful option of adressing the CPU registers indirectly through the Y pointer.

4.1 Serialization and SRAM Use

The greatest size optimization we have implemented is serialization of the algorithm, keeping most of the state in SRAM while we operate on only one byte of state wherever possible. This reduces the instruction count on all parts of the algorithm. It also allows us to make use of fewer dedicated registers. The only procedure where this approach was not successful is the permutation layer, for which we chose to reserve 4 output registers as we believe it allows us to apply the 4-bit period permutation in software with the most size-efficiency.

By requiring only 4 dedicated registers to keep a partial block state, and 6 to keep the rest of the algorithms' state, we were left with exactly 16 of the 32 general purpose registers available to keep the key register (as we rely on the 6 registers for X,Y,Z to navigate the SRAM, indirectly addressed registers and flash storage respectively.) This means support for 128-bit keys was able to be added to the implementation at no extra cost.

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 4 to 7 extra instructions 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 **??** and Table **??**.

The S-box construct replaces a single low nibble in the output. The simplest and most size-efficient way to apply it to a byte in code consists of 1) calling it, 2) swapping the resulting nibbles, 3) calling it again for the other nibble and then 4) swapping the nibbles back. We can call this code starting from step 2 to apply the S-box to only the high nibble of a byte, as is required when scheduling new round keys.

```
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
```

Fig. 3. Loading and unpacking bytes into nibbles in a constant cycle count takes us 8 instructions whereas loading unpacked nibbles takes us only 1. The net gain is only 2 bytes of code.

4.3 Permutation Layer

The code to apply the bit position permutation to the state in software borrows heavily from the AVR implementation of PRESENT drafted in

Louvain by Eisenbarth et al. **?**. Since the permutation follows a 4-bit period in the input, their choice to use 4 bytes of I/O when rotating bits off of registers into corresponding new positions seems quite efficient. In the Louvain implementation one bit is rolled off from 4 state registers into one output register and this block is done twice, completing one output byte.

This implementation of the permutation layer requires availability of 4 bytes of temporary storage for half of the permuted state, which we save to the stack before applying the permutation to the other 4 bytes of the state. The construct in Figure **??** 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. As you can see this takes 4 instructions, whereas a *rcall*, *ret* construction would take us only 3, but this construct doesn't use the stack which means we can keep our intermediate values there rather than requiring 4 more dedicated registers or make more complicated use of the SRAM.

The 4 extra registers that became available through this approach allowed us to implement support for 128-bit keys while keeping the scheduled round keys entirely in CPU registers at no extra size cost. This construct does not affect the cycle count relative to the state or input.

Rather than using specialized code to invert this permutation when decrypting, we use the PRESENT author's design of the bit permutation to undo it by applying it twice more (that is $P(P(P(i))) = i$ for each bit position i).

```
block:
        set                 ; set T flag
        ; fall through
redo_block:
        ; instructions here happen twice when called from block

        brtc continue       ; break if T flag cleared
        clt                 ; clear T flag
        rjmp redo_block     ; redo this block
continue:
        ret
```

Fig. 4. A construct that uses the state register to re-do a block twice with code executing before, after and in between, allowing 2 executions of the block without requiring access to the stack to store return addresses.

4.4 Indirect Register Access

The AVR platform allows the CPU registers to be addressed *indirectly*, meaning we can use a pointer (Y) in memory space to interact with them. We use the feature to load all 10 or 16 of the dedicated key registers in a loop, to iterate over them while applying the round key to the state in SRAM, and to rotate the key registers. This last optimization proved to be devastating to performance compared to inlined rotation of the registers, which is why we added an option to configure either approach.

Doing these operations in a loop which iterates over registers results in smaller code, and allows us to rotate an arbitrary number of registers at a fixed instruction cost. Faster (inlined) rotation makes the implementation about 4 times faster and requires 2/8 extra instructions depending on the configured key size.

4.5 Round Key Application and Key Scheduling

The round key is applied to the state in the SRAM by reading, XORing and writing one byte to/from a register at a time. The use of indirect register access allows us to iterate through the key bytes in CPU registers while iterating the state bytes in SRAM.

When scheduling keys, we 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), as explained in Section **??**.

The inverse key scheduling procedure is only needed in the decryption routine, so we were able to inline it into the decryption round.

4.6 Limits Encountered

Although S-Box application and round key application always happen close to each other and have the same SRAM access pattern, the varying order in which they are applied in encryption/decryption rounds, or the omission of the S-Box application in the final step makes it impossible to combine the two steps into one procedure that makes PRESENT smaller.

It is possible to save a few instructions by iterating through the state in SRAM from wherever the X pointer is located rather than rewinding it to the start of the block in every round procedure of the algorithm. Still, the varying order in which they are applied in the encryption/decryption rounds makes it impossible to do so for the general case in smaller code.

The choice to rewind to the start of the block places the SRAM pointer back to the same address as before encryption or decryption, which seems like a good default for real-life use and also allows all round procedures to be callable from external applications, which seemed more desirable than having them rely on 'hidden' state which affects their behaviour.

4.7 Using the Code for Specific Applications

While the attained size of our implementation of PRESENT should suffice for use in real-world applications, most of the procedure calls can be inlined when only encryption or decryption is required in the application. If only encryption is required, the inverse S-Box and S-Box unpacking code can be omitted as well.

As mentioned in **??**, the key register rotation procedure can be configured to either use indirect register addressing, or be inlined at a cost of 4/16 extra bytes depending on configured key size.

5 Conclusion

We've compared our results to the existing AVR implementation by Eisenbarth et al **?** and the GNU AVR-Crypto-Lib (as a standard C reference) **?**. We are pleased to announce we were able to reduce the code size by 70% and gain 18% speed increase. (Table **??**) 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 992 cycles per encryption (12%). Overall, we managed to push the limits of the PRESENT implementation and establish a wide spectrum of techniques to enable speed and size efficiency.

	Encryption	Decryption	Size
Papagiannopoulos ?	8721	-	1794
AVR Crypto-lib **?**	105796	151624	1514
Eisenbarth **?**	10723	11239	936
Verstegen ?			
Inlined rotation, unpacked S-Boxes (128-bit)	64506	119626	292
Inlined rotation (128-bit)	67854	123346	290
Inlined rotation, unpacked S-Boxes	52622	73952	280
Inlined rotation	55784	77300	278
Unpacked S-Boxes	186883	250032	274
Unpacked S-Boxes (128-bit)	278189	568010	274
Default	190045	253380	272
Default (128-bit)	281537	571730	272

Table 4. Speed (in clock cycles) and size (in bytes) comparisons to existing implementations of PRESENT for the AVR architecture, ordered by size, speed.