PRESENT Cipher Speed Optimizations on ATtiny45

Aram Verstegen , Kostas Papagiannopoulos
Radboud University Nijmegen, Kerckhoff's Institute

March 5, 2013

1 Speed Optimizations

1.1 PRESENT S-Box

In this section we examine the S-Box of the PRESENT cipher from the speed perspective and we view it as a standalone component. We start from the original S-Box, identify its performance issues (section 1.1.1) and we expand it to the more efficient Squared S-Box (section 1.1.2). Based on these improvements, we examine the S-Box and permutation layer combined in section 1.2.1.

1.1.1 Original S-Box

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

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

The core issue with 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 due to the required shifts and XOR operations that need to take place before and after each table lookup.

1.1.2 Squared S-Box

A solution to the afore mentioned problem is to construct a new lookup table that is custom made for the 8-bit AVR architecture. In the following table, we demonstrate a Squared S-Box, which uses an 8-bit input and produces an 8-bit output. As a result, there is no need for overhead computation and the substitution consists of a single lookup.

The Squared S-Box described, is an efficient and viable solution with

x	00	01	02	03	...	0C	0D	0E	0F
S-Box[x]	CC	C5	C6	CB	...	C4	C7	C1	C2
x	10	11	12	13	...	1C	1D	1E	1F
S-Box[x]	5C	55	56	5B	...	54	57	51	52

Table 2: The first two lines of the 256-byte Squared S-Box. It substitutes one byte at a time, without any overhead.

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 also 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 `lpm` instruction (load from FLASH memory - 3 clock cycles). The only processing penalty consists of a 256 byte transfer from FLASH to SRAM which will occur only once, preferably in the beginning or setup phase.

1.2 PRESENT S-Box and Permutation Layer

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, e.g. an n-bit shift requires n clock cycles, and thusly we have to look for alternatives.

1.2.1 Merged SP Lookup Tables

In this direction, work by Peter Schwabe [2] 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. The fastest approach that we identified for the permutation layer is the idea developed by Zheng Gong and Bo Zhu [3, 4]. 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. 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 1.1.2 is only possible in an AVR microcontroller with at least 1024 bytes of internal[2] SRAM, for instance ATtiny1634.

1.2.2 Fast Lookup Table Access

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.

```
LookupTable_i:
mov ZH , high_part_of_address_i ; load ZH with the 8 high bits
mov ZL , low_part_of_address_i  ; load ZL with the table index
lpm register , Z  ; load register with Table[ZL] contents
```

We aim to keep the changes required in `ZH` to a minimum. Thus, we align the four 256-byte tables such that they can be accessed by using only the `ZL` register as an index and keeping `ZH` unchanged. Elaborating, the four lookup tables `LookupTable3, LookupTable2, LookupTable1, LookupTable0` (section 1.2.1) start from `0x0600, 0x0700, 0x0800, 0x0900` respectively and

[1] Benchmarking was performed on ATmega2560.

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

thus, the 8 high bits of the address part (`0x06, 0x07, 0x08, 0x09`) remain the same and the 8 low bits can act as the table index, ranging from 0 to 255 (`0x00` to `0xFF`).
To similar end, we group lookups that search in the same table, as follows.

```
LookupTable_0 ;                 LookupTable_0 ; Group 0
LookupTable_1 ;                 LookupTable_0 ; Group 0
LookupTable_0 ;  transforms to  LookupTable_1 ; Group 1
LookupTable_1 ;                 LookupTable_1 ; Group 1
```

We perform the maximum amount of grouped table lookups, given the limited number of registers. Withing each grouping, `ZH` remains the same.

1.3 Key Update Function

They 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 below.

1.3.1 Key Rotation Operation

The algorithm specifies that the key must be rotated by 61 bits to the left. Given the fact that rotations/shifts are computationally expensive, 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 with the `mov` instructions and the 3 remaining are carried out with the `ror` and `shr` commands.

1.3.2 Key Substitution Operation

The highest 4 bits of the 80-bit key used by PRESENT cipher, must go through the S-Box. To avoid again 4-bit memory access, we craft another Squared S-Box that substitutes the 4 high bits of the 8-bit input, while the low 4 bits remain unchanged. The resulting table can be seen below and the key substitution operation consists of a single lookup.

1.3.3 Key XOR Round Counter 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

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	5	56	5B	...	5C	5D	5E	5F

Table 3: Squared S-Box for key substitution, operating only on the high 4 bits of the input.

need to be XORed with different parts of two separate registers. However, if we perform the XOR operation before the key rotation operation, the bits that will be operated on are bits 34,35,36,37,38 which span a single register (using the previous representation, they are located in register4). Performing this operation before the key rotation does not affect the outcome or security of the algorithm.

1.4 Bitslicing

NOTE: This is not a valid/verified section yet - we only present thoughts on the possibility of bitslicing. To be reconsidered.
Bitslicing has been performed before for PRESENT [5].In order to perform bitslicing in AVR, there is need for a large amount of SRAM, e.g. for a fully-blown bitslicing, we need 8*64=512 bytes of SRAM. With this approach we perform 64 encryptions in bitsliced form and we no longer need the permutation layer. On the downside, there will traffic between registers and SRAM and also, we cannot use lookup tables for the substitution layer and thus, we have to implement the S-Box with boolean functions that operate on each state bit.

References

[1] 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 the proceedings of CHES 2007.

[2] Michael Hutter, Peter Schwabe, *NaCl on 8-bit AVR Microcontrollers*, 2013.

[3] Zheng Gong, Bo Zhu, *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, retrieved 2/2013.

[4] Zheng Gong, Pieter Hartel, Svetla Nikova and Bo Zhu. *Towards Secure and Practical MACs for Body Sensor Networks*, Indocrypt 2009.

[5] Philipp Grabher, Johann Grosschadl and Dan Page, *Light-Weight Instruction Set Extensions for Bit-Sliced Cryptography*