PRESENT Runs Fast:
Efficient and Secure Implementation in Software

Tiago B. S. Reis[1], Diego F. Aranha[1], and Julio López[1]

Institute of Computing – University of Campinas

Abstract. The PRESENT block cipher was one of the first hardware-oriented proposals for implementation in extremely resource-constrained environments. Its design is based on 4-bit S-boxes and a 64-bit permutation, a far from optimal choice to achieve good performance in software. As a result, most software implementations require large lookup tables in order to meet efficiency goals. In this paper, we describe a new portable and efficient software implementation of PRESENT, fully protected against timing attacks. Our implementation uses a novel decomposition of the permutation layer, and bitsliced computation of the S-boxes using optimized Boolean formulas, not requiring lookup tables. The implementations are evaluated in embedded ARM CPUs ranging from microcontrollers to full-featured processors equipped with vector instructions. Timings for our software implementation show a significant performance improvement compared to the numbers from the FELICS benchmarking framework. In particular, encrypting 128 bits using CTR mode takes about 2100 cycles on a Cortex-M3, improving on the best Assembly implementation in FELICS by a factor of 8. Additionally, we present the fastest masked implementation of PRESENT for protection against timing and other side-channel attacks in the scenario we consider, improving on related work by 15%. Hence, we conclude that PRESENT can be remarkably efficient in software if implemented with our techniques, and even compete with a software implementation of AES in terms of latency while offering a much smaller code footprint.

1 Introduction

The need for secure and efficient implementations of cryptography for embedded systems has been an active area of research since at least the birth of public-key cryptography. While considerable progress has been made over the last years, with development of many cryptographic engineering techniques for optimizing and protecting implementations of both symmetric [24] and asymmetric algorithms [9], the emergence of the Internet of Things (IoT) brings new challenges. The concept assumes an extraordinary amount of devices connected to the Internet and among themselves in local networks. Devices range from simple radio-frequency identification (RFID) tags to complex gadgets like smartwatches, home appliances and smartphones; and fulfill a wide variety of roles, from the automation of simple processes to critical tasks such as traffic control and environmental surveillance [5].

In a certain sense, the IoT is already here, as the number of devices storing and exchanging sensitive data rapidly multiplies. Realizing the scale in which security issues arise in this scenario poses challenges in terms of software security, interoperable authentication mechanisms, cryptographic algorithms and protocols. The possible proliferation of weak proprietary standards is particularly worrying, aggravated by the fact that IoT devices are many times physically exposed or widely accessible via the network, which opens up new possibilities of attacks making use of side-channel leakage. These leaks occur through operational aspects of a concrete realization of the cryptographic algorithm, such as the execution time of a program [25, 14]. Consequently, securely implementing cryptographic algorithms in typical IoT devices remains a relevant research problem for the next few years, which is further complicated by the limited availability of resources such as RAM and computational power in these devices.

In order to fulfill the need for cryptographic implementations tailored for resource-constrained embedded devices, many different *lightweight* algorithms have been proposed for various primitives. One such proposal is the PRESENT block cipher [11], a substitution-permutation network designed by Bogdanov *et al.* and published in CHES 2007, that has received a great deal of attention from the cryptologic community and was standardized by ISO for lightweight cryptographic methods [37]. The block cipher has two versions: PRESENT-80 with an 80-bit key, and PRESENT-128 with a 128-bit key, both differing only by the key schedule, being one of its main design goals to optimize the hardware implementation. In this work, we focus on this block cipher, providing an alternative formulation of the original PRESENT algorithm. We discuss why our formulation is expected to be more efficient in software and provide implementation results that support this claim. Also, we analyze the impact of using a second-order masking scheme as a side-channel leakage countermeasure.

Our Contributions. We introduce a new portable and secure software implementation of PRESENT that leads to significant performance improvement compared to previous work. The main idea consists in optimizing the computation of permutation P in two consecutive rounds, by replacing it with two more efficient permutations P_0 and P_1 in alternated rounds. In this work, side-channel resistance is implemented through constant time execution and masking countermeasures. Our implementations are evaluated on embedded ARM processors, but the techniques should remain efficient across platforms. Extensive experimental results provided on both Cortex-M microcontrollers and more powerful Cortex-A processors indicate that we obtained the fastest side-channel resistant implementation of PRESENT for our target architectures.

Organization. Section 2 reviews related work on software implementation of PRESENT and Section 3 describes the original specification of the block cipher. Novel techniques for efficient software implementation are discussed in Section 4, security properties and side-channel countermeasures in Section 5. Section 6 describes our target platforms, relevant aspects about our implementation and present the performance figures we obtained, before comparing them with results from the open research literature. Conclusions are drawn in Section 7.

2 Related Work

The design of PRESENT [11] has motivated an extensive amount of research in the cryptologic community, both in terms of cryptanalysis and engineering aspects. The main results in these regards are summarized here.

Starting from the cryptanalytic results, many techniques have been explored to break PRESENT's security claims [10, 27, 38, 15], and, yet, the best full-round attack found is a biclique attack [27] able to recover the secret key based on $2^{79.76}$ encryptions of PRESENT-80 or $2^{127.91}$ encryptions of PRESENT-128. Although the result is technically a proof that PRESENT is not an ideally secure block cipher, it actually helps building up confidence in the cipher design. After extensive research efforts, the best known attack still requires almost as much computational effort as a brute-force attack.

Regarding the efficient implementation of PRESENT, one of the most comprehensive works is the PhD thesis by Axel Poschmann, one of PRESENT's designers [33]. The author discusses a plethora of implementation results, both in hardware and in software, for a wide selection of architectures, ranging from 4-bit to 64-bit devices. For the software implementations, the author presents different versions optimized for either code size or speed. He focuses on implementing the S-box as a lookup table, which is potentially vulnerable to timing attacks in processors equipped with cache memory. Hence, the optimizations introduced to improve the S-box performance cannot be used in our work, because we are concerned with side-channel security.

In [31], Papapagiannopoulos *et al.* present efficient bitsliced implementations of PRESENT, along with implementations for other block ciphers, having as target architecture the ATtiny family of AVR microcontrollers. This work employs an extension [17] of Boyar-Peralta heuristics [13] to minimize the complexity of digital circuits applied to PRESENT, providing a set of 14 instructions to compute the S-box. Bao *et al.* [6] adapt the approach to implement the inverse S-box in 15 instructions for the LED cipher, which shares the same substitution layer with PRESENT.

Similarly to [31], Benadjila *et al.* [7] also provide bitsliced implementations for many different block ciphers, including PRESENT, but this time for Intel x86 architectures. One of the primary focuses of this work is the usage of SIMD instructions to speed up the implementations through vectorization.

It is also important to cite the work of Dinu *et al.* [18], which implements and optimizes PRESENT alongside with twelve other different block ciphers for three different platforms: 8-bit ATmega, 16-bit MSP430 and 32-bit ARM Cortex-M3. Their best results for PRESENT were obtained through a table-based implementation that merges the permutation layer and the substitution layer of the cipher in some instances. Since the Cortex-M3 is also one of the target architectures for our work, it is relevant to observe actual figures in this case. For this platform, the authors report an execution time of 16,919 clock cycles for encrypting 128 bits of data in CTR mode and 270,603 cycles for running the key schedule, encrypting and decrypting 128 bytes of data in CBC mode.

Out of all the aforementioned works, none of them discusses side-channel security and many even explicitly state the usage of large tables to compute the PRESENT S-box, which is a well-known source of side-channel leakage [12]. However, there are some researchers who address this issue. For example, [22] presents a bitsliced implementation for PRESENT that uses a masking scheme to provide second-order protection against side-channel attacks. The authors use a device endowed with a Cortex-M4 processor and report an execution time of 6,532 cycles to encrypt one 64-bit block, excluding the time consumed by the random number generator in the masking routine. They also provide experimental evidence for the effectiveness of masking as a side-channel attack countermeasure in ARM-based architectures. It is worth noting, however, that the masking scheme used by the authors only aims to protect the S-box computation, hence leaving the key unmasked and the algorithm open to possible attacks that might target specific sections of the code.

At last, we mention the paper [32], which applies a technique called Threshold Implementation to counteract differential power analysis attacks and glitches on hardware circuitry. This alternative masking scheme, originally proposed by Nikova *et al.* [29], has the advantage of not requiring the generation of random bits for computing operations between shares of secret information, but demands the evaluation of multiple S-boxes which can become computationally expensive in software.

3 The PRESENT block cipher

The PRESENT block cipher [11] is a substitution-permutation network (SPN) that encrypts a 64-bit block using a key with 80 or 128 bits. The key is first processed by the key schedule to generate 32 round keys $subkey_1, ..., subkey_{32}$ with 64 bits each. To encrypt a given block of data, it repeats the following steps over 31 rounds: the block is XORed with the corresponding round key; each contiguous set of 4 bits in the block is substituted according to the output of the substitution box (S-box) S; and then the 64 bits are rearranged by a permutation P. After the final round, the block is XORed with $subkey_{32}$. A high-level description of PRESENT encryption is given in Algorithm 1.

The S-box S acts over every 4 bits of the block, as specified in Table 1. Although the most straightforward way to implement the S-box in software is by using a lookup table, [31] shows how to simulate one evaluation of this function by performing 14 Boolean operations over the 4 input bits. Listing 1.1 contains a C-language implementation of the S-box and also of the inverse of this S-box, which can be useful for the decryption algorithm. The S-box was directly obtained from [31] using the extended Boyar-Peralta heuristics [13]. We computed the inverse S-box using the same approach with software from Brian Gladman [21]. Our inverse S-box has 15 instructions and reproduces the same number obtained by Bao *et al.* [6], in which the function was not explicitly given.

Listing 1.1: Bitsliced implementation in C for both the direct and inverse S-boxes of the PRESENT block cipher.

Algorithm 1 PRESENT encryption of one message block.

```
   Input: A 64-bit block of plaintext $B$, a key $K$.
   Output: A 64-bit block of ciphertext $C$.
1: $subkey = (subkey_1, subkey_2, ..., subkey_{32}) \leftarrow keySchedule(K)$
2: $C \leftarrow B$
3: for $i = 1$ to 31 do
4:     $C \leftarrow C \oplus subkey_i$
5:     $C \leftarrow S(C)$
6:     $C \leftarrow P(C)$
7: end for
8: $C \leftarrow C \oplus subkey_{32}$
9: return $C$
```

Table 1: PRESENT S-box, given in hexadecimal notation.

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

```
/* Each macro takes as input 4 words and transforms
 * them in-place according to the S-box function
 * or its inverse.
 */

#define PRESENT_SBOX(x0,x1,x2,x3)       \
    T1 = x2 ^ x1;   T2 = x1 & T1;       \
    T3 = x0 ^ T2;   T5 = x3 ^ T3;       \
    T2 = T1 & T3;   T1 = T1 ^ T5;       \
    T2 = T2 ^ x1;   T4 = x3 | T2;       \
    x2 = T1 ^ T4;   x3 = ~x3;           \
    T2 = T2 ^ x3;   x0 = x2 ^ T2;       \
    T2 = T2 | T1;   x1 = T3 ^ T2;       \
    x3 = T5;

#define PRESENT_INV_SBOX(x0,x1,x2,x3)   \
    T0 = ~x3;       T1 = x2 ^ x0;       \
    T2 = x2 & x0;   T3 = x1 ^ T2;       \
    T4 = x3 ^ T1;   x3 = T0 ^ T3;       \
    T0 = T1 & x3;   T1 = x2 ^ T0;       \
    T2 = T4 | T1;   x0 = T3 ^ T2;       \
    T5 = T4 ^ T1;   T2 = T3 & T5;       \
    x2 = T4 ^ T2;   x1 = T2 ^ (~T1);    \
```

The permutation P is specified by Equation 1 below and moves the i-th bit of the state to the position $P(i)$:

$$P(i) = \begin{cases} 16i \mod 63, & \text{if } i \neq 63, \\ 63, & \text{if } i = 63. \end{cases} \quad (1)$$

From the definition of P, one can easily verify that $P^2 = P^{-1}$. By looking at Figure 1, another interesting property of this permutation can be noticed: if the 64-bit state of the cipher is stored in four 16-bit registers, the application of the permutation P aligns the state in a way that the concatenation of the i-th bit of each of the four registers of the permuted state corresponds to 4 consecutive bits of the original state. These properties will be explored by the technique proposed later.

$$B = \begin{bmatrix} 00\ 01\ 02\ 03\ 04\ 05\ 06\ 07\ 08\ 09\ 10\ 11\ 12\ 13\ 14\ 15 \\ 16\ 17\ 18\ 19\ 20\ 21\ 22\ 23\ 24\ 25\ 26\ 27\ 28\ 29\ 30\ 31 \\ 32\ 33\ 34\ 35\ 36\ 37\ 38\ 39\ 40\ 41\ 42\ 43\ 44\ 45\ 46\ 47 \\ 48\ 49\ 50\ 51\ 52\ 53\ 54\ 55\ 56\ 57\ 58\ 59\ 60\ 61\ 62\ 63 \end{bmatrix},$$

$$P(B) = \begin{bmatrix} 00\ 04\ 08\ 12\ 16\ 20\ 24\ 28\ 32\ 36\ 40\ 44\ 48\ 52\ 56\ 60 \\ 01\ 05\ 09\ 13\ 17\ 21\ 25\ 29\ 33\ 37\ 41\ 45\ 49\ 53\ 57\ 61 \\ 02\ 06\ 10\ 14\ 18\ 22\ 26\ 30\ 34\ 38\ 42\ 46\ 50\ 54\ 58\ 62 \\ 03\ 07\ 11\ 15\ 19\ 23\ 27\ 31\ 35\ 39\ 43\ 47\ 51\ 55\ 59\ 63 \end{bmatrix}.$$

Fig. 1: Matrix representation of the 64-bit input block B and its permutation $P(B)$, both split into four 16-bit rows.

4 Efficient implementation

The main novelty introduced in this work lies in the techniques devised to efficiently implement the PRESENT block cipher in software, which are now described. First, we limit the scope to PRESENT-80, the version using an 80-bit key, which is better suited for lightweight applications due to a smaller memory footprint. The encryption and decryption routines are exactly the same for the 128-bit version, the only difference is in the key schedule, which should not be a critical section of the algorithm in terms of performance. In fact, applying the same techniques exposed here to PRESENT-128, provides, within a 5% margin, the same time measurements for all scenarios we consider.

Algorithm 2 specifies our proposal for implementing encryption of a single block with PRESENT. Essentially, every two applications of permutation P are replaced by evaluations of permutations P_0 and P_1, which satisfy the property that $P_1 \circ P_0 = P^2$, a fact that preserves the correctness of the modified algorithm. The way P_0 and P_1 act upon the cipher state is represented in Figure 2, and code in the C programming language to implement both permutations follows in Listing 1.2. On the description of this algorithm, we use the function S_{BS}, wich we define as being the same S-box used for PRESENT, but taking as inputs state bits whose indexes are congruent modulo 16 instead of every four consecutive bits. In other words, this S-box interprets the state of the cipher as four 16-bit words and operates on them in a bitsliced fashion.

Algorithm 2 Our proposal for PRESENT encryption of one message block.

```
    Input: A 64-bit block of plaintext $B$, a key $K$.
    Output: A 64-bit block of ciphertext $C$.
 1: $subkey = (subkey_1, subkey_2, ..., subkey_{32}) \leftarrow keySchedule(K)$
 2: $C \leftarrow B$
 3: for $i = 1$ to 15 do
 4:     $C \leftarrow C \oplus subkey_{2i-1}$
 5:     $C \leftarrow P_0(C)$
 6:     $C \leftarrow S_{BS}(C)$
 7:     $C \leftarrow P_1(C)$
 8:     $C \leftarrow C \oplus P(subkey_{2i})$
 9:     $C \leftarrow S_{BS}(C)$
10: end for
11: $C \leftarrow C \oplus subkey_{31}$
12: $C \leftarrow P(C)$
13: $C \leftarrow S_{BS}(C)$
14: $C \leftarrow C \oplus subkey_{32}$
15: return $C$
```

We need to observe two facts to prove the equivalence between Algorithm 1 and Algorithm 2. First, the S-box S in Algorithm 1 acts on the same quadruplets of bits that S_{BS} acts on Algorithm 2, since both P_0 and P bitslice the state over 16-bit words. Then note that $P(P(X \oplus subkey_i) \oplus subkey_{i+1}) = P^2(X \oplus subkey_i) \oplus P(subkey_{i+1}) = P_1(P_0(X \oplus subkey_i)) \oplus P(subkey_{i+1})$, being that leftmost term exactly the transformation undergone by state X over rounds i and $i+1$ on Algorithm 1 and the rightmost term the transformation undergone by state X over rounds i and $i+1$, for i odd, on Algorithm 2, when we disregard the S-box step on both algorithms. Since the S-boxes operate equivalently and, without S-boxes, the algorithms are also equivalent, the proof is concluded.

Now, at first glance, it may not be clear why our alternative version for PRESENT is faster than the original one, but there are two main advantages. The first one is due to complexity in software. Permutations P_0 and P_1 are simply more software friendly, requiring less operations to be implemented, when compared to the permutation P. An evidence to corroborate this fact was obtained from the source code generator for bit permutations provided by Jasper Neumann in [28], estimating a cost of 14 clock cycles to execute either P_0 or P_1 and a cost of 24 cycles to execute P, when implemented optimally.

Listing 1.2: Efficient implementation in C of the permutations P_0 and P_1 of our proposal for PRESENT encryption.

```
/* The following macros permute two 64-bit blocks
 * simultaneously, using an auxiliary variable t
 * and storing one block on the high 16-bit word
 * of the 32-bit variables X0, X1, X2 and X3, and
 * the other block on the low 16-bit word of the
 * same variables.
 */

#define PRESENT_PERMUTATION_P0(X0,X1,X2,X3) \
    t = (X0^(X1>>1)) & 0x55555555;           \
    X0 = X0^t; X1 = X1^(t<<1);               \
    t = (X2^(X3>>1)) & 0x55555555;           \
    X2 = X2^t; X3 = X3^(t<<1);               \
    t = (X0^(X2>>2)) & 0x33333333;           \
    X0 = X0^t; X2 = X2^(t<<2);               \
    t = (X1^(X3>>2)) & 0x33333333;           \
    X1 = X1^t; X3 = X3^(t<<2);               \

#define PRESENT_PERMUTATION_P1(X0,X1,X2,X3) \
    t = (X0^(X1>>4)) & 0x0F0F0F0F;           \
    X0 = X0^t; X1 = X1^(t<<4);               \
    t = (X2^(X3>>4)) & 0x0F0F0F0F;           \
    X2 = X2^t; X3 = X3^(t<<4);               \
    t = (X0^(X2>>8)) & 0x00FF00FF;           \
    X0 = X0^t; X2 = X2^(t<<8);               \
    t = (X1^(X3>>8)) & 0x00FF00FF;           \
    X1 = X1^t; X3 = X3^(t<<8);               \
```

$$B = \begin{bmatrix} 00 & 01 & 02 & 03 & 04 & 05 & 06 & 07 & 08 & 09 & 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\ 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\ 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \end{bmatrix},$$

$$P_0(B) = \begin{bmatrix} 00 & 16 & 32 & 48 & 04 & 20 & 36 & 52 & 08 & 24 & 40 & 56 & 12 & 28 & 44 & 60 \\ 01 & 17 & 33 & 49 & 05 & 21 & 37 & 53 & 09 & 25 & 41 & 57 & 13 & 29 & 45 & 61 \\ 02 & 18 & 34 & 50 & 06 & 22 & 38 & 54 & 10 & 26 & 42 & 58 & 14 & 30 & 46 & 62 \\ 03 & 19 & 35 & 51 & 07 & 23 & 39 & 55 & 11 & 27 & 43 & 59 & 15 & 31 & 47 & 63 \end{bmatrix},$$

$$P_1(B) = \begin{bmatrix} 00 & 01 & 02 & 03 & 16 & 17 & 18 & 19 & 32 & 33 & 34 & 35 & 48 & 49 & 50 & 51 \\ 04 & 05 & 06 & 07 & 20 & 21 & 22 & 23 & 36 & 37 & 38 & 39 & 52 & 53 & 54 & 55 \\ 08 & 09 & 10 & 11 & 24 & 25 & 26 & 27 & 40 & 41 & 42 & 43 & 56 & 57 & 58 & 59 \\ 12 & 13 & 14 & 15 & 28 & 29 & 30 & 31 & 44 & 45 & 46 & 47 & 60 & 61 & 62 & 63 \end{bmatrix}.$$

Fig. 2: Matrix representation of the 64-bit input block B and its permutations $P_0(B)$ and $P_1(B)$, all of them divided into four 16-bit rows.

The second advantage of our proposal involves the application of the S-box. A careful analysis of Algorithm 2 leads to the conclusion that, at lines 6, 9 and 13, where the S-box is applied, the state of the variable C is not the same as the state to which the S-box is applied in Algorithm 1. At line 6, the state has undergone an extra P_0 permutation in relation to the original formulation; and at lines 9 and 13 the state has undergone an extra evaluation of P. By looking at Figures 1 and 2, it stands clear that, if the ciphertext is stored into four 16-bit registers, both P and P_0 organize the state in such a way that every four consecutive bits are aligned in columns throughout those four registers, similarly to what would be seen in a fully bitsliced implementation. Therefore, an implementation following the structure in Algorithm 2 can make use of bitwise operations to simulate the S-box step, calculating sixteen S-box applications simultaneously.

The same rationale may be applied to generate other alternative versions of the PRESENT encryption algorithm. Figure 3 illustrates different versions of PRESENT obtained by interchanging S-box applications and permutations. In this figure, S represents the S-box applied over every four consecutive bits of the state and S_{BS} represents the S-box computed in a bitsliced fashion.

One last observation to further improve performance of the implementation in a 32-bit architecture is that two blocks of plaintext can be encrypted in CTR mode at once, organizing the state such that 32 S-boxes are calculated simultaneously instead of only 16. For a 64-bit architecture, the same strategy can be carried out to encrypt four blocks at once.

All of the algorithmic observations and implementation techniques discussed here extend directly to the decryption routine, as shown by Algorithm 3. The inversion of encryption is particularly simplified by the fact that P_0 and P_1 are involutory permutations, that is, $P_0^{-1} = P_0$ and $P_1^{-1} = P_1$. The involutory property of P_0 and P_1 has yet another advantage. Since $P_1 \circ P_0 = P^2$ and $P^2 = P^{-1}$, it follows that $P = P_0 \circ P_1$, what might be used to reduce the code size of the implementation, because the permutation P does not need to be implemented provided that P_0 and P_1 have already been coded.

Algorithm 3 Our proposal for PRESENT decryption of one message block.

```
    Input: A 64-bit block of ciphertext $C$, a key $K$.
    Output: A 64-bit block of plaintext $B$.
 1: $subkey = (subkey_1, subkey_2, ..., subkey_{32}) \leftarrow keySchedule(K)$
 2: $B \leftarrow C$
 3: $B \leftarrow B \oplus subkey_{32}$
 4: $B \leftarrow S_{BS}^{-1}(B)$
 5: $B \leftarrow P^{-1}(B)$
 6: $B \leftarrow B \oplus subkey_{31}$
 7: for $i = 15$ to $1$ do
 8:     $B \leftarrow S_{BS}^{-1}(B)$
 9:     $B \leftarrow B \oplus subkey_{2i}$
10:     $B \leftarrow P_1(B)$
11:     $B \leftarrow S_{BS}^{-1}(B)$
12:     $B \leftarrow P_0(B)$
13:     $B \leftarrow B \oplus P(subkey_{2i-1})$
14: end for
15: return $B$
```

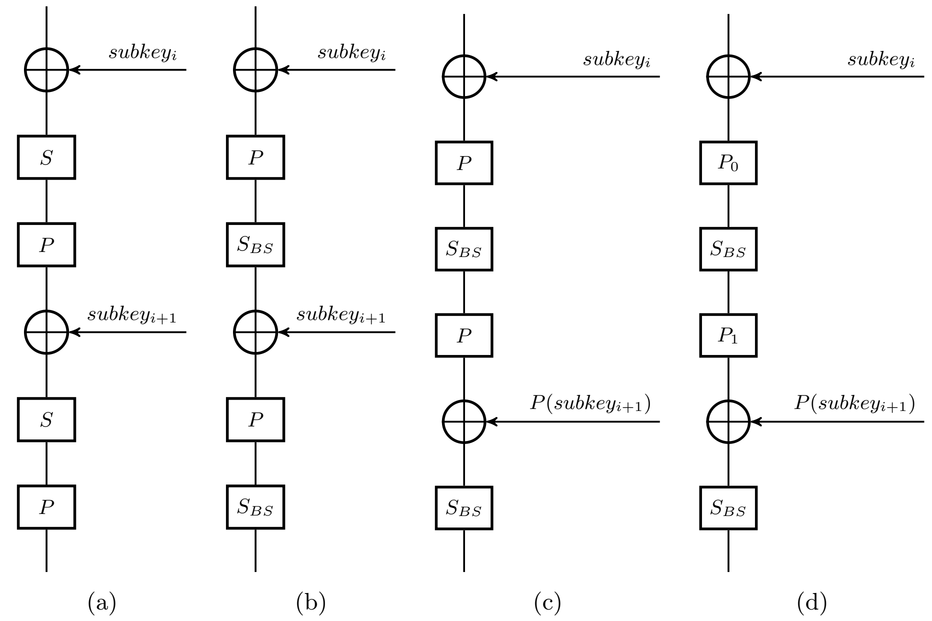

Fig. 3: Diagram showing equivalent ways to implement two consecutive rounds (i and $i+1$) of PRESENT for encryption. The original specification for the block cipher corresponds to the leftmost diagram and the rightmost one corresponds to the version proposed here with alternating P_0 and P_1 permutations.

At last, it is important to notice that our proposal has the drawback of applying the permutation P to some of the round keys. Although, since typically many blocks of message are encrypted or decrypted with the same key, the key schedule routine should have a low impact on the algorithm's practical performance, since it is executed only once for several executions of encryption/decryption routines.

5 Side-channel countermeasures

As commented previously, there has been extensive work on the cryptanalysis of PRESENT and the lack of significant advances provides evidence that the cipher is likely to fulfill the desired security goals. However, even if the block cipher design is ideally secure, a careless implementation may leak sensitive data during execution and undermine the security of the algorithm with its insecure realization.

Particularly, a major concern is side-channel attacks, that is, attacks which are crafted based on information obtained from the physical implementation of a cryptographic primitive. For instance, an attacker may gather data such as execution time of an algorithm [25, 14, 19], power consumption [30], sound produced by the hardware [20] or even magnetic radiation emitted during the computation [26] and, through these data, the attacker may gain access to sensitive information processed by the device under analysis.

It is worth noting that side-channel attacks are limited to situations where the attacker has physical access to the hardware executing the implementation or at least can interact with the device through the network. It is not completely unreasonable to ignore the possibility of such attacks when the implementation of the algorithm is physically protected from the attacker or not accessible for any kind of interaction, but reality tends to go in the opposite direction in the IoT context. In this scenario, devices are frequently accessible to the attacker by either physical means or through the network and typically lack tamper-resistance countermeasures for protecting the hardware from external influence.

5.1 Protecting against timing attacks

The focus is primarily on timing attacks, since they are entirely within the scope of software implementation, and appear to be the most practical side-channel attack. Furthermore, protecting software implementations from more invasive side-channel attacks is very challenging, since the software countermeasures can be typically circumvented by an invasive attacker. Recent work has developed static analysis tools to detect variances in execution time correlated with secret information at a rather low level [34, 2], allowing implementers to formally guarantee constant execution time of their code or at least implement mitigations.

In practice, the main sources of timing vulnerabilities are memory accesses and conditional branches depending on secret data. Conditional branching, by definition, may cause different instructions to be executed among different runs of a program, which, by its turn, may cause the execution time of the algorithm to depend on sensitive data given as input. The effect of branch misprediction in more sophisticated processors may further interfere with pipelined datapaths and provoke significant variations [1]. In a similar way, if a processor is equipped with cache memory, the execution time may leak information about the rate of cache misses or hits during memory accesses, and, clearly, if these accesses depend on sensitive data, this implementation becomes susceptible to side-channel attacks [8]. Therefore, by avoiding these situations, a software implementation can encrypt a message block in constant time, independently of characteristics about the inputs (plaintext message or cryptographic key). This runtime property is called *isochronicity*.

5.2 Masking the implementation

Ensuring that code runs in constant time is sufficient to render timing attacks impractical, although other side-channel leakages might still be exploited. Another family of techniques for improving side-channel resistance is called *secret sharing*, or *masking*, which consists in splitting sensitive variables occurring in the computation into $d + 1$ shares (or masks) in order to unlink the correlation between environmental information and the secret data being processed.

A masking technique based on $d+1$ masks is said to be a d-th order masking and can only be broken by an attacker who manages to obtain leakage related to at least $d+1$ intermediate variables of the algorithm. It is possible to prove that the difficulty for a side-channel attack to succeed, in practice, increases exponentially with d and, hence, the masking order can be considered a sound criterion to evaluate the robustness of the implementation against side-channel analysis [16].

The literature presents different alternatives to implement a masked encryption algorithm [32], but analysis will be restricted to the proposal given by Ishai *et al.* in [23], which appears to be the most appropriate for a fast software implementation. In this proposal, the masked state of a sensitive variable m with $d+1$ *shares* is

$$m = \bigoplus_{i=0}^{d} m_i = m_0 \oplus m_1 \oplus \ldots \oplus m_d, \tag{2}$$

where each m_i is a share of the secret and all shares form together a masked secret. In order to create a masked implementation on the variable m, one can randomly generate the d masks $m_1, m_2, ..., m_d$ and calculate m_0 such that Equation 2 holds.

From this definition, we can derive ways to calculate different operations over the masks. The following list contains all operations necessary to implement a masked version of PRESENT.

1. A `NOT` operation over a masked secret has to be carried out as a `NOT` operation performed on an odd number of masks to preserve the relationship in Equation 2. A single mask can just as well be negated:

$$\neg m = \neg m_0 \oplus m_1 \oplus \ldots \oplus m_d.$$

2. An `XOR` operation between masked secrets $a = \bigoplus_{i=0}^{d} a_i$ and $b = \bigoplus_{i=0}^{d} b_i$ can be performed by calculating the `XOR` of all corresponding masks:

$$a \oplus b = \bigoplus_{i=0}^{d} (a_i \oplus b_i).$$

3. An `AND` operation between two masked secrets is more complicated and can be computed as follows: for every pair (i, j), $1 \leq i < j \leq d+1$, generate a random bit $z_{i,j}$. Then, compute $z_{i,j} = (z_{i,j} \oplus a_i b_i) \oplus a_j b_i$. Now, for every $1 \leq i \leq d+1$, the i-th share may be computed as

$$m_i = a_i b_i \oplus \bigoplus_{i \neq j} z_{i,j}.$$

4. An `OR` operation might be calculated using the logical identity $\texttt{OR}(a, b) = \neg(\neg a \cdot \neg b)$, which depends only on operations previously defined.

The nonlinear operations `OR` and `AND` stand out as the most expensive ones, requiring $O(d^2)$ calls to a random bit generator and memory to store a matrix z of $O(d^2)$ entries. This is the main drawback of the technique in resource-constrained devices and makes the use of high-order masking impractical in many scenarios.

6 Implementation details and results

6.1 Target architecture

Currently, there is a vast variety of processors under consideration for integration to the IoT. The focus given in this work is on some representatives of the ARM architecture, since it is the world leader in the market of microprocessors and, thus, attracts relevant academic work as well as commercial interest. More specifically, our implementations were benchmarked on the following platforms:

– **Cortex-M0+:** Arduino Zero powered by an Atmel SAMD21G18A ARM Cortex-M0+ CPU, clocked at 48MHz.
– **Cortex-M3:** Arduino Due powered by an Atmel SAM3X8E ARM Cortex-M3 CPU, clocked at 84MHz.
– **Cortex-M4:** Teensy 3.2 board containing a MK20DX256VLH7 Cortex-M4 CPU, clocked at 72 MHz.
– **Cortex-A7/A15:** ODROID-XU4 board containing a Samsung Exynos5422 2GHz Cortex-A15 and Cortex-A7 octa-core CPU.
– **Cortex-A53:** ODROID-C2 board containing an Amlogic 64-bit ARM 2GHz Cortex-A53 (ARMv8) quad-core CPU.

Members of the Cortex-M [4] family are commonly used in embedded applications, being found on devices ranging from medical instrumentation equipment to domestic household appliances. The design of these processors is optimized for cost and energy efficiency, making them relatively low-end when compared to the other targets.

As for the members of Cortex-A [3] family, they are more computationally powerful than the Cortex-M processors, being able to execute complex tasks such as running a robust operating system or a high-quality multimedia task. These processors have access to the NEON engine, a powerful Single Instruction Multiple Data (SIMD) extension, and may have sophisticated out-of-order execution.

6.2 Main results

In order to discuss our results, the code size and speed of our implementations are measured in two scenarios based on what is proposed in the FELICS framework [18], such that results can be comparable in a fair and reliable manner.

Scenario 1 simulates a communication protocol established in sensor networks or between IoT devices. It is assumed here that the device possesses the master key stored in RAM, calculates the key schedule and then proceeds to encrypt and decrypt 128 bytes of sensitive data using the CBC mode of operation. Due to the employment of the CBC mode, the suggested trick of encrypting more than one block in parallel does not work, since this mode of operation forces dependencies between consecutive input blocks. Hence, it stands clear that it is not the optimal scenario to use our techniques, but we still chose to implement it exactly as described in [18] for the sake of comparison.

Scenario 2 simulates an authentication protocol in which the block cipher is used to encrypt 128 bits of data in CTR mode of operation. The round keys are assumed to be stored in memory and, consequently, no key schedule is required. This is a very appropriate stance to employ all of the optimizations proposed so far, since the CTR mode encrypts and decrypts blocks of input independently.

Results for both scenarios are expressed in Table 2 and Table 3. All the measurements were based on code fully written in C language, compiled by GCC 6.3.1 in the case of the Cortex-A family and by GCC 4.8.4 for the Cortex-M family, using the flag `-O3` for optimized speed results. The isochronicity property of the constant time implementations was validated using the FlowTracker static analysis tool [34]. FlowTracker performs information flow analysis from function inputs marked as secret to branch instructions and memory addresses, effectively detecting and thwarting timing attacks. This tool analyzes compiled code at the LLVM Intermediate Representation level, thus closer to the platform-specific native code. All timings for Cortex-M processors were reproduced to a reasonable degree in the ARM Cortex-M Prototyping System (MPS2), an FPGA-based board with support to microcontrollers ranging from the Cortex-M0 to M7. However, we only report timings collected in the widely available platforms to simplify comparisons with future competing implementation efforts.

One of the main observations attained from these measurements is that the cost to protect the implementations with masking is high, especially in lower-end processors. In our case, a second-order masking was used and the time consumed by the random number generator was disregarded. Still, a slowdown of up to 6.8 times was observed in the case of the Cortex-M0+. For higher-end processors, however, the slowdown can be inferior to a 4-factor. Throughout all processors, a sensible increase in code size due to masking is observed.

Another fact to notice is that, as expected, even when differences in input size are taken into account, the performance of PRESENT in Scenario 2 is substantially better than the performance in Scenario 1, mainly due to the choice of mode of operation. In Scenario 1, using the CBC mode, only decryption can be parallelized, and encryption ends up being roughly twice as slow as in CTR mode.

6.3 Vector implementation using NEON

For the platforms with access to NEON instructions, parallelism within the PRESENT encryption algorithm can also be explored for enhancing performance. In particular, it is relevant to mention that the NEON instructions `VTBL` and `VTBX` allow the computation of fast table lookups by performing register operations, without the need of memory accesses.

Table 2: Performance results for Scenario 1 – key schedule, encryption and decryption of 128 bytes in CBC mode – of side-channel resistant implementations of PRESENT, encompassing both isochronous (constant time) and second-order masking countermeasures.

Processor	Code size [bytes]	Key schedule [cycles]	Encryption [cycles]	Decryption [cycles]
Isochronous implementation				
Cortex-M0+	1436	6381	46429	23445
Cortex-M3	1320	5043	29442	16291
Cortex-M4	1328	3464	22993	11731
Cortex-A7	2732	3232	21027	10657
Cortex-A15	1792	1740	14780	7050
Cortex-A53	2484	1554	13583	3726
Masked implementation				
Cortex-M0+	8056	7145	332079	204690
Cortex-M3	7048	4628	197601	122521
Cortex-M4	9216	3413	186556	100417
Cortex-A7	9248	2657	116004	64041
Cortex-A15	9248	1894	59474	29130
Cortex-A53	8452	1943	39983	12848

Besides the original formulation of the algorithm, that implements S-boxes as lookup tables, we were also able to evaluate the performance of a different proposal mentioned in [33] and attributed to Gregor Leander. The idea is similar to ours, in principle, since it decomposes the permutation P into two others. However, Leander's decomposition aims to allow a faster lookup table-based implementation, which is the opposite direction we are looking for. Still, even using the NEON instructions to implement the lookup tables used in Leander's method, our formulation was found to be faster.

NEON implementations can process eight blocks simultaneously due to the support of 128-bit registers, in the same fashion as processing two blocks in parallel in 32-bit processors or four blocks in parallel using 64-bit ones. For this reason, neither scenario used previously is appropriate to evaluate vector implementations. Scenario 1 does not support parallelism due to the mode of operation employed and Scenario 2 processes only 128 bits of data, which is only two blocks of input, not making use of the full capacity of processing eight blocks at once.

For this reason, we chose to analyze the performance of our NEON implementations under a third scenario, in which we run the key schedule, encrypt and decrypt 128 bytes of data. These results are reported in Table 4 and Table 5, alongside with the results of the native implementation, without vector instructions, to provide a baseline for comparison.

Table 3: Performance results for Scenario 2 – encryption of 128 bits in CTR mode – of side-channel resistant implementations of PRESENT, encompassing both isochronous (constant time) and second-order masking countermeasures.

Processor	Code size [bytes]	Execution time [cycles]
Isochronous implementation		
Cortex-M0+	2524	3183
Cortex-M3	2476	2116
Cortex-M4	2612	1599
Cortex-A7	2456	1708
Cortex-A15	2456	960
Cortex-A53	2536	1052
Masked implementation		
Cortex-M0+	12392	21744
Cortex-M3	9728	12387
Cortex-M4	11012	11096
Cortex-A7	13322	7482
Cortex-A15	13322	3688
Cortex-A53	18028	3681

Table 4: Performance results for isochronous execution of the key schedule, encryption and decryption of 128 bytes of data in CTR mode, using both serial and vectorized code.

Processor	Code size [bytes]	Key schedule [cycles]	Encryption+Decryption [cycles]
Serial implementation			
Cortex-M0+	2524	6381	47884
Cortex-M3	2476	5043	31830
Cortex-M4	2612	3464	26785
Cortex-A7	2456	2732	27161
Cortex-A15	2456	1740	14169
Cortex-A53	2536	1554	7406
Vector implementation using NEON			
Cortex-A7	2798	2299	14274
Cortex-A15	2798	1533	8083
Cortex-A53	3908	1552	7322

By analyzing the results, we notice that the NEON instructions were able to provide a meaningful speedup for the 32-bit processors. For the 64-bit Cortex-A53, however, the efficiency of native instructions associated with the possibility

Table 5: Performance results for execution of the key schedule, encryption and decryption of 128 bytes of data in CTR mode, using both serial and vectorized code, protected by second-order masking.

Processor	Code size [bytes]	Key schedule [cycles]	Encryption+Decryption [cycles]
Serial implementation			
Cortex-M0+	12392	7145	345619
Cortex-M3	9728	4628	205244
Cortex-M4	11012	3413	192454
Cortex-A7	13322	2657	119542
Cortex-A15	13322	1894	58635
Cortex-A53	18028	1943	23207
Vector implementation using NEON			
Cortex-A7	2798	2671	76286
Cortex-A15	2798	1948	28633
Cortex-A53	3908	1941	28343

of processing four blocks in parallel beats the vector implementation by a small margin. Naturally, these implementations have a substantial impact on code size when compared to Table 2.

Notice also that the only difference introduced by this third scenario compared to Scenario 1 is the choice of the mode of operation. It further illustrates how much better CTR performs in this case, in which we can make use of the parallelism intrinsic to the encryption routine.

6.4 Comparison with related work

Although many implementation results for PRESENT are published, we focus here on comparing our metrics to the works of [18] and [22], which are, to the best of our knowledge, the most efficient publicly available implementations of PRESENT in similar platforms to the ones we use.

In [18], a series of implementations is presented for many block ciphers which are benchmarked on a Cortex-M3 processor. For a scenario identical to the Scenario 2 we described, they report an execution time of 16,786 clock cycles and a code size of 3,568 bytes. Our results are almost 8 times better considering the execution time, and over 30% better regarding the code size. They also measure these metrics for Scenario 1, in which they report the usage of 270,603 cycles of execution and 2,528 bytes of code, which is slower and more space-consuming than our implementation, but by a smaller margin, since the CBC mode of operation employed in this case does not benefit from some of the optimizations.

The work of [22] showcases a bitsliced implementation of PRESENT on a Cortex-M4, protected by a second-order masking. It claims to be able to encrypt one input block in 6,532 cycles. We argue that our results are better, since, even if there is no penalty caused by the tight coupling with a mode of operation, it would encrypt 128 bits of data in 13,064 cycles, which is slower than the 11,096 cycles we achieved for the same processor on Scenario 2. Furthermore, since this implementation has a bitslice factor of 32, it cannot actually encrypt only 128 bits of data without having to do extra work, whereas our implementation is not only faster, but more flexible in the sense that it allows small amounts of data to be efficiently encrypted.

It is also relevant to take into consideration performance results from other block ciphers to gauge how useful our techniques may be in practice. In particular, we take a closer look at AES, arguably the most extensively used block cipher today and which has been originally praised for its good performance in software [35]. The current state-of-the-art implementations for AES on Cortex-M processor are from [36], in which several different results are presented. Table 6 compares our results to theirs when encrypting 128 bits of data through CTR mode in constant time. We notice that PRESENT is slower than AES on Cortex-M3, but slightly faster on Cortex-M4 and, on both processors, PRESENT's code footprint is several times smaller.

Table 6: Comparison between our results for PRESENT and results from [36] for AES when encrypting 128 bits of data in CTR mode, in constant time.

Implementation	Code size [bytes]	Execution time [bytes]
AES on Cortex-M3	12120	1617
PRESENT on Cortex-M3	2476	2116
AES on Cortex-M4	12120	1618
PRESENT on Cortex-M4	2612	1599

7 Conclusion

In this work, we presented a novel technique for accelerating encryption and decryption using the PRESENT block cipher. Our modified algorithm is expected to be faster in software when compared to the original PRESENT specification for many platforms and, indeed, our experimental data supports that we were able to significantly outperform state-of-the-art results for processors within the ARM Cortex-M family. This makes PRESENT competitively efficient even when compared to secure implementations of widely used software-oriented ciphers such as AES.

Furthermore, our proposal has the advantage to be readily implemented in constant time, which is relevant in contexts where there is concern regarding side-channel attacks. For further side-channel security, we implemented and analyzed the performance impact of a second-order masking scheme.

At last, we show that our technique can also be applied to vector implementations – using the ARM-NEON extension, for example – to achieve even higher performance gains in some compatible platforms.

8 Acknowledgements

The authors gratefully acknowledge financial support from LG Electronics Inc. during the development of this research, under the project "*Efficient and Secure Cryptography for IoT*". The third author also acknowledges financial support from CNPq: a research productivity scholarship.

References

1. Aciiçmez, O., Koç, c.K., Seifert, J.P.: On the power of simple branch prediction analysis. In: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security. pp. 312–320. ASIACCS '07, ACM, New York, NY, USA (2007), `http://doi.acm.org/10.1145/1229285.1266999`
2. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations (2016)
3. ARM: Cortex-A Series Family. `https://www.arm.com/products/processors/cortex-a/index.php`, accessed: June, 2016
4. ARM: Cortex-M Series Family. `https://www.arm.com/products/processors/cortex-m/index.php`, accessed: June, 2016
5. Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. Computer Networks 54(15), 2787–2805 (2010), `https://doi.org/10.1016/j.comnet.2010.05.010`
6. Bao, Z., Luo, P., Lin, D.: Bitsliced implementations of the prince, LED and RECTANGLE block ciphers on AVR 8-bit microcontrollers. In: ICICS. Lecture Notes in Computer Science, vol. 9543, pp. 18–36. Springer (2015)
7. Benadjila, R., Guo, J., Lomné, V., Peyrin, T.: Implementing lightweight block ciphers on x86 architectures. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 8282, pp. 324–351. Springer (2013)
8. Bernstein, D.J.: Cache-timing attacks on aes. `\tthttp://cr.yp.to/papers.html\#cachetiming` (2005)
9. Bernstein, D.J.: Curve25519: New diffie-hellman speed records. In: Public Key Cryptography. Lecture Notes in Computer Science, vol. 3958, pp. 207–228. Springer (2006)
10. Blondeau, C., Nyberg, K.: Links between truncated differential and multidimensional linear properties of block ciphers and underlying attack complexities. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 8441, pp. 165–182. Springer (2014)
11. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: CHES. Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer (2007)

12. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: CHES. Lecture Notes in Computer Science, vol. 4249, pp. 201–215. Springer (2006)
13. Boyar, J., Peralta, R.: A new combinational logic minimization technique with applications to cryptology. In: SEA. Lecture Notes in Computer Science, vol. 6049, pp. 178–189. Springer (2010)
14. Cheval, V., Cortier, V.: Timing attacks in security protocols: Symbolic framework and proof techniques. In: POST. Lecture Notes in Computer Science, vol. 9036, pp. 280–299. Springer (2015)
15. Cho, J.Y.: Linear cryptanalysis of reduced-round PRESENT. In: CT-RSA. Lecture Notes in Computer Science, vol. 5985, pp. 302–317. Springer (2010)
16. Coron, J., Prouff, E., Rivain, M., Roche, T.: Higher-order side channel security and mask refreshing. In: FSE. Lecture Notes in Computer Science, vol. 8424, pp. 410–424. Springer (2013)
17. Courtois, N., Hulme, D., Mourouzis, T.: Solving circuit optimisation problems in cryptography and cryptanalysis. IACR Cryptology ePrint Archive 2011, 475 (2011), `http://eprint.iacr.org/2011/475`
18. Dinu, D., Corre, Y.L., Khovratovich, D., Perrin, L., Großschädl, J., Biryukov, A.: Triathlon of lightweight block ciphers for the internet of things. IACR Cryptology ePrint Archive 2015, 209 (2015), `http://eprint.iacr.org/2015/209`
19. Doychev, G., Köpf, B.: Rational protection against timing attacks. In: Fournet, C., Hicks, M.W., Viganò, L. (eds.) IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015. pp. 526–536. IEEE (2015), `http://dx.doi.org/10.1109/CSF.2015.39`
20. Genkin, D., Shamir, A., Tromer, E.: RSA key extraction via low-bandwidth acoustic cryptanalysis. In: CRYPTO (1). Lecture Notes in Computer Science, vol. 8616, pp. 444–461. Springer (2014)
21. Gladman, B.: Serpent S Boxes as Boolean Functions. `http://www.gladman.me.uk/`
22. de Groot, W., Papagiannopoulos, K., de la Piedra, A., Schneider, E., Batina, L.: Bitsliced masking and ARM: friends or foes? In: LightSec. Lecture Notes in Computer Science, vol. 10098, pp. 91–109. Springer (2016)
23. Ishai, Y., Sahai, A., Wagner, D.A.: Private circuits: Securing hardware against probing attacks. In: CRYPTO. Lecture Notes in Computer Science, vol. 2729, pp. 463–481. Springer (2003)
24. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: CHES. Lecture Notes in Computer Science, vol. 5747, pp. 1–17. Springer (2009)
25. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: CRYPTO. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996)
26. Kuhn, M.G.: Electromagnetic eavesdropping risks of flat-panel displays. In: Privacy Enhancing Technologies. Lecture Notes in Computer Science, vol. 3424, pp. 88–107. Springer (2004)
27. Lee, C.: Biclique cryptanalysis of PRESENT-80 and PRESENT-128. The Journal of Supercomputing 70(1), 95–103 (2014), `http://dx.doi.org/10.1007/s11227-014-1103-3`
28. Neumann, J.: Code generator for bit permutations. `http://programming.sirrida.de/calcperm.php`
29. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: ICICS. Lecture Notes in Computer Science, vol. 4307, pp. 529–545. Springer (2006)
30. O'Flynn, C., Chen, Z.D.: Side channel power analysis of an AES-256 bootloader. In: CCECE. pp. 750–755. IEEE (2015)

31. Papapagiannopoulos, K.: High Throughput in Slices: The Case of PRESENT, PRINCE and KATAN64 Ciphers. In: RFIDSec. Lecture Notes in Computer Science, vol. 8651, pp. 137–155. Springer (2014)
32. Poschmann, A., Moradi, A., Khoo, K., Lim, C., Wang, H., Ling, S.: Side-channel resistant crypto for less than 2, 300 GE. J. Cryptology 24(2), 322–345 (2011), https://doi.org/10.1007/s00145-010-9086-6
33. Poschmann, A.Y.: Lightweight cryptography: cryptographic engineering for a pervasive world. Ph.D. thesis, Ruhr University Bochum (2009), http://d-nb.info/996578153
34. Rodrigues, B., Pereira, F.M.Q., Aranha, D.F.: Sparse representation of implicit flows with applications to side-channel detection. In: CC. pp. 110–120. ACM (2016)
35. Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N., Kohno, T., Stay, M.: The Twofish Team's Final Comments on AES Selection. https://www.schneier.com/academic/paperfiles/paper-twofish-final.pdf, accessed: March, 2017
36. Schwabe, P., Stoffelen, K.: All the AES you need on Cortex-M3 and M4. In: Avanzi, R., Heys, H. (eds.) Selected Areas in Cryptology – SAC 2016. Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg (2016), to appear.
37. for Standardization, I.O.: ISO/IEC 29192-2:2012. https://www.iso.org/standard/56552.html, accessed: February, 2017
38. Wang, M.: Differential cryptanalysis of PRESENT. IACR Cryptology ePrint Archive 2007, 408 (2007), http://eprint.iacr.org/2007/408