**Noname manuscript No.**
(will be inserted by the editor)

# Efficient and secure software implementations of Fantomas

**Rafael J. Cruz · Antonio Guimarães · Diego F. Aranha**

the date of receipt and acceptance should be inserted later

**Abstract** In this paper, the efficient software implementation and side-channel resistance of the LS-Design construction is studied through a series of software implementations of the Fantomas block cipher, one of its most prominent instantiations. Target platforms include resource-constrained ARM devices like the Cortex-M3 and M4, and more powerful processors such as the ARM Cortex-A15 and modern Intel platforms. The implementations span a broad range of characteristics: 32-bit and 64-bit versions, unprotected and side-channel resistant, and vectorized code for NEON and SSE instruction sets. Our results improve the state of the art substantially, both in terms of efficiency and compactness, by making use of novel algorithmic techniques and features specific to the target platform. We finish by proposing and prototyping instruction set extensions to reduce by half the performance penalty of the introduced side-channel countermeasures.

**Keywords** LS-Design · Fantomas · side-channel resistance · vectorization · instruction set extension.

## 1 Introduction

Efficient cryptography for embedded systems has been a very active field of research for a few decades, and it recently gained renewed interest with the emergence of the Internet of Things, under the moniker *lightweight cryptography*. Applications of cryptography can indeed solve problems faced by connected devices collecting and exchanging sensitive information through an open network. Typical solutions involve authenticated encryption for data protection in transit or at rest, and code signing for secure firmware updates.

Significant interest has been dedicated to the design and implementation of block ciphers, since they represent a fundamental primitive from which many security properties in the symmetric setting can be provided. In that direction, many innovative block ciphers were proposed to maximize performance in resource-constrained devices and to provide lighter but secure alternatives to AES [DR02]. Remarkable examples are PRESENT [BKL+07], PRINCE [BCG+12], and more recently SPARX [DPU+16]. These lightweight designs follow and combine multiple constructions, such as Feistel, Substitution-Permutation and ARX networks, posing distinct trade-offs in terms of efficiency, compactness and resistance against different types of attacks.

Even when cryptographic algorithms can be considered secure according to the latest theoretical cryptanalytic results, their corresponding implementations may be susceptible to attacks based on information leakage. Side-channel analysis is a growing and important issue for cryptographic security, especially in embedded systems where devices are physically accessible to an attacker. These attacks are based on information leaked during computation through side channels such as execution time [Koc96], power consumption [KJJ99], acoustic and electromagnetic emanations. When successful, they help the adversary to identify and recover secret data from observations captured during execution, overcoming the much higher computational cost of cryptanalysis or exhaustive search in the key space. Secret data may be a long-term private key, an ephemeral

Rafael J. Cruz and Antonio Guimarães
Institute of Computing, University of Campinas, Campinas, Brazil
E-mail: {raju@lasca, antonio.junior@students}.ic.unicamp.br

Diego F. Aranha
Department of Engineering, Aarhus University, Aarhus, Denmark
E-mail: dfaranha@eng.au.dk

session key or partial information about the internal state of a primitive, including bits of the plaintext or round keys. The attacks may be based on a small number of observations, such as Branch Prediction [AKS07] or Simple Power Attacks (SPA); or require traces from many consecutive observations, as in the case of Differential Power Attacks (DPA) [KJJ99]. Resistance to side-channel attacks has been considered as a strong security requirement for modern ciphers, and algorithms which facilitate addition of side-channel countermeasures have been thus favored in the scientific literature, bringing attention to ciphers like PICARO [PRC12] and Fantomas [GLSV14].

The LS-Design paradigm [GLSV14] was created with side-channel resistance in mind, because it allows the designer to construct lightweight algorithms friendly to implementation of side-channel countermeasures. LS-Design ciphers typically combine a bitsliced substitution layer with a linear diffusion layer implemented with precomputed tables, both amenable to masking schemes. Masking was initially proposed in 2003 [ISW03] in the context of protecting circuits against probing, but the technique has been later extended to much more complex operations, achieving provable security guarantees [RP10]. Masked implementations have the interesting property that the entire computation is performed over *shared secrets*, decorrelating any potential side-channel leakages from the actual data being encrypted or the real cryptographic keys. From this point of view, masking can be seen as a collection of data perturbation techniques to introduce external noise in the encryption or decryption processes, acting as countermeasure against several types of side-channel attacks.

**Our contributions.** This work extends our previous work [CA16] and presents several efficient, compact, portable and secure (in the sense of side-channel resistant) implementations of the Fantomas block cipher:

- In terms of performance, a number of implementation techniques are described to save execution time or code, several of them easily adaptable to other LS-Designs, such as the CAESAR second-round candidate SCREAMv3 [GLS$^+$15b]. The techniques include a simple and efficient representation of the internal state, an efficient way to organize state in vectorized implementations, and strategies for exploiting parallelism in the CTR mode of operation. Our unprotected 32-bit implementation achieves performance improvements ranging from 3.9% to 62.6% in the ARM Cortex-M architecture, while consuming considerably less code. The vector implementations naturally provide much higher throughput, especially if 16 blocks can be processed simultaneously.
- In terms of security, timing attacks on LS-Designs are discussed together with cache protection heuristics, constant time execution (*isochronicity*), and masking as countermeasures. Cache protection heuristics attempt to reduce effectiveness of cache-timing attacks [Ber04, BM06]; isochronous implementation avoids vulnerable precomputed tables to protect execution against timing attacks; and masking protects against generic side-channel attacks but with a significant performance penalty, illustrating several challenges to future research. The constant time property of the isochronous implementation was validated by hand at the source code level and through static and dynamic analysis tools.
- Instruction set extensions for flexible parity computation are proposed to reduce the performance penalty of the isochronous implementation, while preserving resistance against timing attacks. The extensions were prototyped in an Altera NIOS II platform synthesized on an FPGA and reduced the performance penalty by half.

This paper is organized as follows. Section 2 introduces LS-Designs and the Fantomas block cipher. Section 3 discusses cache-timing attacks and countermeasures for LS-Designs. Section 4 summarizes some characteristics of the target platforms and discusses multiple implementations of Fantomas, targeting ARM and Intel instruction sets. Section 5 presents experimental results and Section 6 details instruction set extensions to reduce the performance impact of side-channel countermeasures. Section 7 concludes the paper.

## 2 LS-Designs and Fantomas

The LS-Design construction is a framework for designing lightweight block ciphers while addressing threats posed by side-channel attacks. Instances of an LS-Design cipher are characterized by the choice of bitsliced S-boxes $S$, an L-box matrix $L$ acting as the diffusion layer, a number of rounds $N_r$ and corresponding round constants $Const$. A particular feature of LS-Designs is the lack of a complex or even any key schedule, saving on storage for temporary variables. In the original LS-Design paper, two algorithms were instantiated and analyzed: Robin, a faster involutive instance that later succumbed to invariant subspace attacks [LMR15]; and the non-involutive cipher Fantomas. Algorithm 1 shows a generic specification for an LS-Design, illustrating its simplicity and regularity. For Fantomas, parameters are $s = 8$ and $l = 16$, resulting in a cipher with a 128-bit key length and 128-bit block size.

**Algorithm 1** LS-Design construction encrypting plaintext block $P$ with key $K$ to generate a ciphertext block $C$.

$X \leftarrow P \oplus K$ ▷ State $X$ represents an $s \times l$-bit matrix
**for** $0 \leq r < N_r$ **do**
  **for** $0 \leq i < l$ **do** ▷ S-box layer with bitslicing
    $X[i, \star] = S[X[i, \star]]$
  **end for**
  **for** $0 \leq j < s$ **do** ▷ L-box layer with table lookups
    $X[\star, j] = L[X[\star, j]]$
  **end for**
  $X \leftarrow X \oplus K$ ▷ Key addition
  $X \leftarrow X \oplus Const(r)$ ▷ Addition of round constants
**end for**
$C \leftarrow X$
**return** $C$

Fantomas employs 3-round 3/5-bit S-boxes with similar structure to the MISTY cipher [CDL15], as presented in detail on Algorithm 2. An important consideration taken by the designers of the cipher is the number of nonlinear operations in the choice of S-boxes. Because Fantomas employs 8-bit S-boxes, they must contain at least 8 nonlinear operations to not be weak from a cryptanalytic point of view. There is some security margin in this design decision because Fantomas employs 11 `AND` operations between elements of the cipher state. However, the additional `AND`s penalize the masking countermeasure, as later discussed in Section 3. The L-box presented in Figure 1 provides diffusion and its computation can be seen as a sequence of vector-matrix products in $\mathbb{F}_2$, as illustrated in the picture.

**Algorithm 2** MISTY-like 3/5 bits S-boxes operating over state $x = \{X_0, X_1, \ldots, X_7\}$.

▷ S5
$X_2 \leftarrow X_2 \oplus (X_0 \wedge X_1)$
$X_1 \leftarrow X_1 \oplus X_2$
$X_3 \leftarrow X_3 \oplus (X_0 \wedge X_4)$
$X_2 \leftarrow X_2 \oplus X_3$
$X_0 \leftarrow X_0 \oplus (X_1 \wedge X_3)$
$X_4 \leftarrow X_4 \oplus X_1$
$X_1 \leftarrow X_1 \oplus (X_2 \wedge X_4)$
$X_1 \leftarrow X_1 \oplus X_0$
▷ Extend-Xor
$X_0 \leftarrow X_0 \oplus X_5$
$X_1 \leftarrow X_1 \oplus X_6$
$X_2 \leftarrow X_2 \oplus X_7$
▷ Key
$X_3 \leftarrow \neg X_3$
$X_4 \leftarrow \neg X_4$
▷ S3: 3-bit Keccak S-box
$t_0 \leftarrow X_5$
$t_1 \leftarrow X_6$
$t_2 \leftarrow X_7$
$X_5 \leftarrow X_5 \oplus ((\neg t_1) \wedge t_2)$
$X_6 \leftarrow X_6 \oplus ((\neg t_2) \wedge t_0)$
$X_7 \leftarrow X_7 \oplus ((\neg t_0) \wedge t_1)$
▷ Truncate-Xor
$X_5 \leftarrow X_5 \oplus X_0$
$X_6 \leftarrow X_6 \oplus X_1$
$X_7 \leftarrow X_7 \oplus X_2$
▷ S5
$X_2 \leftarrow X_2 \oplus (X_0 \wedge X_1)$
$X_1 \leftarrow X_1 \oplus X_2$
$X_3 \leftarrow X_3 \oplus (X_0 \wedge X_4)$
$X_2 \leftarrow X_2 \oplus X_3$
$X_0 \leftarrow X_0 \oplus (X_1 \wedge X_3)$
$X_4 \leftarrow X_4 \oplus X_1$
$X_1 \leftarrow X_1 \oplus (X_2 \wedge X_4)$
$X_1 \leftarrow X_1 \oplus X_0$

## 3 Side-channel security

In this section, we discuss the concept of masking for protecting implementations against side-channel attacks and how to implement this countermeasure for different types of operations recurrent in block ciphers. We also discuss a cache-timing attack against LS-Designs to motivate the countermeasures we later propose.

### 3.1 Masking scheme

Masking is one of the most investigated countermeasures against side-channel cryptanalysis, in particular against different variants of power analysis. In the context of block ciphers, masking aims to protect sensitive data, such as plaintext during encryption or intermediate values during decryption. Because information computed in these processes will be later transformed into the algorithm outputs, intermediary states must be protected at all times. The masked state of $m$ with $d+1$ *shared secrets* is given by $m_0 \oplus m_1 \oplus \ldots \oplus m_d = \bigoplus_{i=0}^{d} m_i = m$, where each $m_i$ is a shared secret and all shared secrets form together a *masked secret*. From this definition, we can collect some observations that allow any cryptographic algorithm to be implemented in a masked way.

1. Applying a linear operation over a masked secret $m$ is equivalent to applying the same operation over shared secrets of $m$:

$$\begin{aligned} L(m) &\equiv L(m_0 \oplus m_1 \oplus \ldots \oplus m_d) \\ &\equiv L(m_0) \oplus L(m_1) \oplus \ldots \oplus L(m_d) \end{aligned}$$

2. A `NOT` operation over a masked secret $m$ can be computed as:

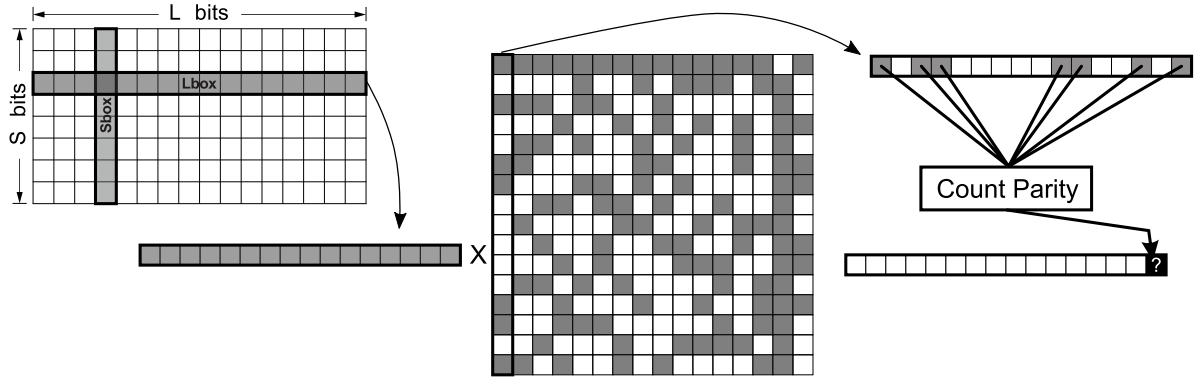
$$\neg m \equiv \neg m_0 \oplus m_1 \oplus \ldots \oplus m_d.$$

3. A `XOR` operation between masked secrets $a = \bigoplus_{i=0}^{d} a_i$ and $b = \bigoplus_{i=0}^{d} b_i$ can be seen as:

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

4. An `AND` operation between two masked secrets $a = \bigoplus_{i=0}^{d} a_i$ and $b = \bigoplus_{i=0}^{d} b_i$ is more complicated and can be computed with the help of Algorithm 3.

Fig. 1: Linear layer of Fantomas, with gray cells representing `1` and white cells `0` values in the L-box matrix. The L-box computation can be seen as a matrix multiplication over $\mathbb{F}_2$, where each row of the cipher state is multiplied by a column of the matrix and the resulting bit is the parity of their bitwise product.




**Algorithm 3** Nonlinear `AND` operation performed on two masked secrets $a$ and $b$ [ISW03].

**Require:** Shares $(a_i)$ and $(b_i)$ satisfying $\oplus_{i=0}^{d} a_i = a$ and $\oplus_{i=0}^{d} b_i = b$.
**Ensure:** Shares $(c_i)$ satisfying $\oplus_{i=0}^{d} c_i = a \wedge b$
1: **for** $i$ **from** 0 **to** $d$ **do**
2: $r_{i,i} \leftarrow 0$;
3: **for** $j$ **from** $i+1$ **to** $d$ **do**
4: $r_{i,j} \leftarrow random()$;
5: $r_{j,i} \leftarrow (r_{i,j} \oplus (a_i \wedge b_j)) \oplus (a_j \wedge b_i)$;
6: **end for**
7: **end for**
8: **for** $i$ **from** 0 **to** $d$ **do**
9: $c_i \leftarrow a_i \wedge b_i$;
10: **for** $j$ from 0 to $d$ **do**
11: $c_i \leftarrow c_i \oplus r_{i,j}$;
12: **end for**
13: **end for**

Line 4 of Algorithm 3 presents an important challenge in terms of performance, since fresh random numbers must be generated for each distinct computation to achieve provable security guarantees. By considering that every share $a_i$ of $a$ represents a *unity*, every masked `AND` computation requires $(d+1)^2 - \frac{d+1}{2}$ units of random data and additional space of $(d+1)^2$ to store a matrix containing all possible combinations of shares. In practice, randomness recycling and other heuristics are often used to reduce the performance penalty incurred by masking strategies, with potential impact on security [BGG+14]. A recent work reduced the time complexity, allowing masking to scale efficiently to higher orders [JS17].

### 3.2 Attacks and countermeasures on LS-Designs

The L-boxes in the LS-Design construction present an obstacle for their secure implementation. Because matrix multiplication in $\mathbb{F}_2$ involves many individual bit operations which turn out to be computationally expensive in software, computation of the diffusion layer may become critical to performance, and the L-box computation is thus commonly implemented through cheaper table lookups. However, table lookups using secret indexes are vulnerable to cache-timing attacks [Ber04, BM06], more recently through Flush+Reload attacks and variants [YF14].

Cache-timing attacks allow an attacker to recover critical data in form of plaintext/key bits or portions of the internal state. While the S-boxes in LS-Designs can be secured against cache-timing attacks when implemented with bitslicing, this attack methodology can be easily extended to the L-Boxes implemented with table lookups. In Line 7 of Algorithm 1, values of the internal state are used to index the L-box. If an adversary is able to monitor the latency of each individual memory access, for instance by means of a Flush+Reload attack, the complete internal state $X$ of the cipher at a given round can be recovered. A successful strategy consists in recovering the internal state just before the last key addition in the last round of encryption, or after the first key addition in the first round of decryption. If this happens for a ciphertext block $C$ later transmitted and captured over the network, the key $K$ can be directly computed with $K = x \oplus C \oplus Const(N_r - 1)$.

There are several possibilities for countermeasures to protect the L-box computation: isochronicity, cache protection heuristics and masking. Implementing the L-box computation as an explicit matrix product by removing secret indexes and precomputed tables results in an isochronous implementation with uniform memory access pattern and response latency. However, the performance impact is substantial, since all the bitwise computations in the L-box must now be performed *on-*

*line*. Another possibility involves heuristics to guarantee that the entire L-box table is always in cache, by either employing smaller tables or visiting all cache lines at every L-box access. Conventional masking schemes can also be applied to introduce noise to make cache-timing attacks more difficult, but if the L-box is computed over all shares of $x$ and key addition is only performed in the first share, the attacker only needs to recover the first share of the internal state to compute the key $K$. Thus the key $K$ must also be decomposed in a set of additive shares as an improved countermeasure, forcing the attacker to recover all shares of the internal state to derive the key $K$. While this is not sufficient to fully overcome the threat of cache-timing attacks, at least the effort dedicated by the attacker is closer to what is expected in the threat model for masking. Algorithm 4 formalizes the masking scheme, by replacing the S-boxes with a masked version and computing the remaining steps over shares of the internal state and key. The next section explores the implementation of these countermeasures in detail.

**Algorithm 4** LS-Design implemented with masking to encrypt $d+1$ shares of plaintext $P \coloneqq \{p_0, \ldots, p_d\}$ and key $K \coloneqq \{k_0, \ldots, k_d\}$ resulting in the ciphertext $c$.

**for** $0 \leq \ell \leq d$ **do** ▷ Initialize the internal state
  $x_\ell = p_\ell \oplus k_\ell$
**end for**
**for** $0 \leq r < N_r$ **do**
  $x = MaskedS(x)$ ▷ Masked S-box layer
  **for** $0 \leq \ell \leq d$ **do** ▷ L-box layer
    **for** $0 \leq j < s$ **do**
      $x_\ell[\star, j] = L[x_\ell[\star, j]]$
    **end for**
  **end for**
  **for** $0 \leq \ell \leq d$ **do** ▷ Add masked key to shares
    $x_\ell = x_\ell \oplus k_\ell$
  **end for**
  $x_0 = x_0 \oplus \mathrm{Const}(r)$ ▷ Add round constants
**end for**
$c \leftarrow \bigoplus_{\ell=0}^{d} x_\ell$
**return** $c$

## 4 Implementation

In this section, we present the multiple implementations of Fantomas, aiming at performance and side-channel security targets. We discuss portable implementations for 32-bit and 64-bit processors implemented in the C programming language, mostly targeting ARM platforms, and additional code vectorized for SSE/NEON instructions. Strategies for masked implementation are discussed last, before experimental results are presented in the next section.

### 4.1 The target platforms

The ARM Cortex-M is a set of 32-bit ARM processor cores intended for microcontroller use, composed of the Cortex-M0, M0+, M1, M3, M4, and M7. These microcontrollers implement load-store architectures optimized for embedded systems in low-power applications. The Cortex-M processors implement slightly different subsets of the more restricted Thumb and Thumb-2 instruction sets, tailored to small code size. With the exception of the M7, Cortex-M microcontrollers do not have internal cache memory, but it is possible to integrate a system-level cache. The Cortex-A family of processors is tailored for more time-consuming applications and provide sophisticated out-of-order execution and NEON vector instruction sets. Cortex-A processors typically have large amounts of cache memory, which can be disabled only in privileged mode. The register file has 16 general purpose registers (`r0-r15`), although pointer arithmetic is restricted to the lower half. A distinctive feature of ARM processors is the possibility to apply a bitwise operation to a second operand of an arithmetic instruction by means of a built-in barrel shifter.

The Intel platform is well-known for its aggressive out-of-order execution and rich vector instruction set. The Streaming SIMD Extensions (SSE) support many vector operations over integers or floating point values, many of them useful for fast cryptography, such as the byte shuffle instruction `PSHUFB`, also available in ARM NEON under the `VTBL` mnemonic. Byte shuffling instructions take 128-bit registers filled with bytes $r_a = a_0, a_1, \ldots, a_{16}$ and $r_b = b_0, b_1, \ldots, b_{15}$ and replace $r_a$ with the permutation $a_{b_0}, a_{b_1}, \ldots, a_{b_{15}}$. A powerful use of this instruction is to perform 16 simultaneous lookups in a 16-byte lookup table, computing a mapping from 4-bit sets to 8-bit values. This can be easily done by storing the lookup table in $r_a$ and the lookup indexes in $r_b$.

### 4.2 Unprotected 32/64-bit implementations

The description starts from the unprotected 32-bit implementation, realized exclusively in the C programming language. Fantomas requires S/L-boxes which operate over 16-bit chunks and other operations over 32-bit data, such as key addition. Therefore, a portable and efficient implementation must simultaneously support the two data types in one concise structure to rep-

resent the internal state. Following the C99 standard, breaking strict aliasing pointer rules can be prevented by representing the internal state as a union combining pointers to the data types, as in Listing 1. This allows the compiler to have sufficient information to optimize arithmetic and memory accesses for both 16- and 32-bit chunks without introducing explicit type conversions and the risk of interference with neighboring data chunks. In 32-bit mode, the internal state is represented as a vector of 4 objects of this type. The interface of the encryption and decryption functions do not have to be modified and still take word-aligned byte vectors as input and conveniently convert them to 32-bit pointers when needed.

Listing 1: Internal state is represented using a vector of unions to respect strict aliasing, such that two different pointers cannot reference the same memory address.

```
typedef union {
  uint32_t u32;
  uint16_t u16[2];
} U32_t;
```

The substitution layer is computed using the union structure. Some operations over 16-bit chunks in the bitsliced S-boxes could be combined in 32-bit operations to increase arithmetic density, but this was avoided to prevent unaligned loads and stores which cause performance degradation. For the linear diffusion layer, the unprotected variable-time version employs two 256-position half-word precomputed tables. A small code portion illustrating the unprotected L-box can be found in Listing 2, where `st` stores the 128-bit state, `LBoxH` transforms the 8 most significant bits and `LboxL` transforms the 8 less significant bits for all $j \in \{0, 1, 2, 3\}$.

Listing 2: Unprotected L-Box using the 16-bit values of the internal state. `LBoxH` maps the higher 8 bits of the linear transformation and `LBoxL` the lower 8 bits.

```
st[j].u16[0] = LBoxH[st[j].u16[0]>>8] ^
               LBoxL[st[j].u16[0] & 0xff];
st[j].u16[1] = LBoxH[st[j].u16[1]>>8] ^
               LBoxL[st[j].u16[1] & 0xff];
```

To improve performance slightly, the key addition works by accumulating the key in the internal state using 32-bit XOR operations, as in Listing 3.

Listing 3: Key addition of Fantomas using the 32-bit state of the union.

```
for(j=0; j<4; j++) {
  st[j].u32 ^= key_32[j];
}
```

The portable 64-bit implementations are a generalization of the 32-bit implementations and mostly follow the same structure. The internal state is represented using a different union, to allow simultaneous operations over 16-bit and 64-bit data chunks without violating strict aliasing rules, as specified in Listing 4. The S-boxes must again be implemented over the union without breaking alignment and causing performance penalties. The unprotected L-box follows the same structure as the corresponding 32-bit implementation.

Listing 4: Union type representing part of the internal state for 64-bit platforms.

```
typedef union {
  uint64_t u64;
  uint16_t u16[4];
} U64_t;
```

### 4.3 Cache protection heuristics

Two versions were implemented with mitigations against cache-timing attacks: a compact one storing the entire L-box in a single cache line and a cache-filling implementation that visits all L-box cache lines at every access.

*Compact implementation*

In this version the S-boxes still need to follow the implementation of the union without breaking alignment. The L-box is represented in four small tables, so the mapping of the L-box changes from 16 bits to 4 bits. This way, each table will contain 32 bytes, fitting a single cache line in modern processors (ARM and Intel), given that the compiler is instructed to align the base address properly. Because every table is used only once in the L-box computation, memory access patterns do not reveal critical information.

A piece of code illustrating the idea can be seen in Listing 5, where `state` stores the 128-bit state, `LBHH` maps the highest 4 bits of the linear transformation, `LBHL` maps the 4 least significant bits of the most significant byte, `LBLH` maps the 4 most significant bits of the least significant byte and `LBLL` the remaining least significant 4 bits, for all $j \in \{0, 2, 4, 6, 8, 10, 12, 14\}$.

*Cache-filling implementation*

This version is similar to the cache-protected versions, but it always brings to cache memory the two entire 256-position half-word precomputed tables by visiting multiple cache lines.

Listing 5: Compact implementation using four small tables to store the entire L-box in a single cache line.

```
uint8_t *b = (uint8_t *)st;
...
for (j=0, k=0; j<4; j+=2, k+=2) {
  st[ j ].u16[0] = LBLH[b[k]>>4] ^ LBLL[b[k] & 0xf] ^ LBHH[b[k+1]>>4] ^ LBHL[b[k+1] & 0xf];
  k+=2;

  st[ j ].u16[1] = LBLH[b[k]>>4] ^ LBLL[b[k] & 0xf] ^ LBHH[b[k+1]>>4] ^ LBHL[b[k+1] & 0xf];
  k+=2;

  st[j+1].u16[0] = LBLH[b[k]>>4] ^ LBLL[b[k] & 0xf] ^ LBHH[b[k+1]>>4] ^ LBHL[b[k+1] & 0xf];
  k+=2;

  st[j+1].u16[1] = LBLH[b[k]>>4] ^ LBLL[b[k] & 0xf] ^ LBHH[b[k+1]>>4] ^ LBHL[b[k+1] & 0xf];
}
```

Listing 6: `LBoxH` contains the upper part of the linear transformation and `LBoxL` contains the lower part, `CPU_CACHELINE` contains the cache line size (default is 64). The loop traverses all the cache lines where the table will be stored.

```
uint16_t LBoxH[256] __attribute__ ((aligned (CPU_CACHELINE))) = { ... };
uint16_t LBoxL[256] __attribute__ ((aligned (CPU_CACHELINE))) = { ... };
uint16_t tmp;
...
for(j=0; j<(256/CPU_CACHELINE); j++) {
  tmp ^= LBoxL[j*CPU_CACHELINE];
  tmp ^= LBoxH[j*CPU_CACHELINE];
}
...
st[0].u16[0] ^= tmp;
...
st[0].u16[0] ^= tmp;
```

It is necessary to explicitly force that precomputed tables start at a cache-line boundary, which may not be compatible with compiler defaults. The code to access the whole table can be seen in the Listing 6. In the code, `CPU_CACHELINE` contains the size of the cache line and the tables are aligned according to the size of the cache line, by default `CPU_CACHELINE` $= 64$ as commonly found in ARM and Intel processors. The loop scans the table by shifting in cache lines to bring the entire table to cache memory before performing the linear transformation.

### 4.4 Isochronous 32/64-bit implementation

The isochronous implementations are a little more involved. The S-box layer implemented through bitslicing fortunately provides isochronicity already, so no additional countermeasures are needed. The diffusion layer is performance-critical and presents more obstacles to side-channel resistance, since it is usually implemented through table lookups on the L-box. The protected version implements the operation online by performing vector-matrix binary multiplications, where two 16-bit chunks are processed at the same time. The code portion in Listing 7 illustrates part of it, where `x` stores the 32 bits to be transformed by the L-box in 16-bit pairs and `y` contains the `l`-th duplicate line of the binary matrix representing the linear transformation. This function computes the dot product of the two 32-bit vectors in $\mathbb{F}_2$, and calculates the parity of each 16-bit result, processing two transformations at the same time. Function `ProdLBox` was transformed to operate over 64 bits with simple modifications to the input and output types and a repeated bit mask `0x0001000100010001` in the last operation, allowing computation of 4 simultaneous evaluations of the L-box (Listing 8).

Listing 7: Multiply the $s$-th row of the matrix $L$ containing the value $y = (y_s, y_s)$ by the value $x = (x_a, x_b)$ where the result is the $s$-th value of $(x \cdot L)_s = (x_a \cdot y_s, x_b \cdot y_s)$.

```
static inline uint32_t ProdLBox(uint32_t x,
    uint32_t y, uint8_t s) {
  x &= y;
  x ^= x >> 8;
  x ^= x >> 4;
  x ^= x >> 2;
  x ^= x >> 1;
  return (x & 0x00010001) << s;
}
```

Listing 8: Explicit L-box computation, adapted to 64-bit platforms.

```
static inline uint64_t ProdLBox(uint64_t x,
    uint64_t y, uint8_t s) {
  x &= y;
  x ^= x >> 8;
  x ^= x >> 4;
  x ^= x >> 2;
  x ^= x >> 1;
  return (x & 0x0001000100010001) << s;
}
```

*Platform-specific 64-bit implementation*

As a possibly faster alternate option for Intel platforms, we also implemented a 64-bit version using the `POPCNT` instruction for *population counting*. This instruction is part of SSE4 extension and counts the number of `1` bits over a 64-bit register, storing the parity in the least significant bit. Because the instruction takes 3 cycles to complete [Fog16] and only a single bit of the result is useful for our computation, this version performed much less efficiently. For reference, it can be found in Listing 9 below.

Listing 9: L-box evaluation with parity computation computed through the least significant bit of the result from Intel `POPCNT` instruction.

```
static inline uint64_t ProdLBox(U64_t x,
    uint64_t y, uint8_t s) {
  x.u64 &= y;
  return ((_mm_popcnt_u16(x.u16[0])&0x1) ^
  ((_mm_popcnt_u16(x.u16[1]) & 0x1)<<16) ^
  ((_mm_popcnt_u16(x.u16[2]) & 0x1)<<32) ^
  ((_mm_popcnt_u16(x.u16[3]) & 0x1)<<48))
  << s;
}
```

### 4.5 Masked implementation

The masked implementation needs large modifications in the S-boxes, because every operation computed in Algorithm 2 must now be replaced by the operations specified in Section 3.1. Others modifications in the algorithm are described in Algorithm 4.

Countermeasures in the linear layer are still needed during encryption and decryption to protect against the cache-timing attack discussed in Section 3.2, because the linear layer comes immediately before the last key addition in the encryption and immediately after the first key addition in the decryption. If successful, the cache-timing attack would disclose the internal state in these positions and, with knowledge of the ciphertext, an attacker could mount a critical key recovery attack.

Two functions are essential for preprocessing the blocks before masked encryption and decryption can be performed. These functions convert a plaintext block to a masked block and the converse, respectively. The first function must generate $d$ randomized blocks and combine these blocks with the original by means of `XOR` operations to generate the last block. The second function must combine all masked blocks with `XOR` operations after encryption and decryption are processed.

A substantial amount of random bits is required to generate the masked blocks and to compute the masked `AND` described in Algorithm 3. Random number generation was implemented through the standardized Hash_DRBG [BK12] algorithm instantiated with the SHA-256 hash function. This choice proved to be faster than reading bytes from `/dev/urandom` by a 10-factor in the Linux-enabled platforms. Even with this faster option, generating random bits still imposes a massive performance penalty and dominates the execution time in our masked implementation. Because this is highly platform-specific, we take two approaches: follow related work and exclude the random generation time from the experimental results, and also measure the time for random number generation for comparison.

### 4.6 Further exploiting parallelism

Although the S-box implementation already extracts some internal parallelism inside Fantomas, we further note that there is much more room for exploiting parallelism. Under a mode of operation amenable to parallelization of both encryption and decryption such as CTR, Fantomas can be implemented quite efficiently at the cost of flexibility and code size by processing multiple blocks simultaneously. In particular, the 32-bit implementation can be adapted to process 2 blocks at the

same time, by sharing each 32-bit word in the internal state among two 16-bit chunks of two consecutive blocks; and the 64-bit implementation can be adapted in a similar way to process 4 consecutive blocks. This optimization also has the effect of accelerating the S-box computation, because internal horizontal dependencies between 16-bit chunks from the same block are now eliminated and the S-boxes can be computed with 32-bit operations alone, without introducing unaligned memory accesses or other performance penalties. Because the CBC mode imposes a serialization of encryption, vectorized implementations should stick to the CTR mode processing multiple blocks simultaneously, where only the counters are encrypted/decrypted and later added to the plaintext/ciphertext, respectively. For comparison, we also implemented a single-block vectorized Fantomas in CBC mode.

*Vectorized implementation*

Our implementations target the ARM and Intel platforms equipped with modern vector instruction sets capable of computing high-throughput bitwise operations over full vector registers, and performing fast lookups over small precomputed tables. LS-Designs become very friendly to vectorization under these conditions.

Recall that the internal state of Fantomas is represented as 8 pieces of 16 bits each, hereby called *lanes* to follow vectorization terminology. The S-boxes are computed in a bitsliced way, facilitating vectorization as long as the S-layer can compute over at least 8 blocks simultaneously, applying the same operation over each 16-bit lane from the same block. The L-box presents a higher obstacle, because memory accesses should be reduced to increase arithmetic density. There are two clear ways of implementing the L-box with higher arithmetic density: the first one is to perform an explicit vector-matrix multiplication over $\mathbb{F}_2$ as in the constant-time 32/64-bit implementation; or employing byte shuffling instructions for table lookups inside vector registers. In the latter, registers are sliced in byte-sized chunks, processing 16 blocks simultaneously, where the individual bytes can be stored and transposed in a matrix to guarantee that every vector register has the same $i$-th byte of each block. These two approaches were implemented and the latter was clearly faster due to higher occupancy of the vector registers. For portability over Intel and ARM, the table lookups were implemented using the GCC intrinsic `__builtin_shuffle()` for byte shuffling, which translates to the instructions `PSHUFB` and `VTBL` discussed on Section 4.1.

Since the L-box is a linear transformation, the 16 bits can be broken in smaller pieces. The L-box in Figure 2 can be split in 4-bit lanes and the table reduced to 4 tables of 16 positions storing 16-bit values. To make use of the table lookup instructions mapping 4-bit values to 8 bits, the splitting must divide the most significant bytes from the least significant bytes and the entire table is stored in 8 vector registers of 128 bits. Listing 10 presents part of the vectorized linear layer.

The single-block CBC version operates separately in the most significant bytes and least significant bytes, and combines them together at the end. The 16-block CTR version is a little more complex and follows the organization adopted by the vectorized implementation of SCREAMv3 [GLS+15b]. First, it is necessary to expand the CTR counter for the 16 simultaneous blocks. After expansion, the counters must be transposed and stored in a different order. Counter updates can be done by propagating carries using vector comparisons. The expanded counter is computed from the original counter as in Figure 2a, and the state must be partially transposed and stored as in Figure 2b. Partial transposition is not too computationally expensive, because the organization required is an intermediate step of the transposition algorithm.

The organization in Figure 2 must be kept through the whole process, because then the substitution layer can be performed in the first 8 blocks and then on the final 8 blocks. The linear layer is similar to the single-block version and the splitting is not required, since the least significant bytes are stored in the first 8 blocks in Figure 2b and the most significant bytes in the remaining 8 blocks in the same Figure. The key must also be transformed in a similar way as in Figure 2c to facilitate the key addition step. A total of 16 copies of the key are stored in a set of registers and partially transposed to match the organization used for the counter, such that the correct operands are used for all the additions. Our SSE-vectorized implementations of Fantomas are publicly available for independent benchmarking and reproducibility[1].

## 5 Experimental results

Our implementations were benchmarked in seven different ARM and Intel platforms. The compiler used for the Cortex-M platforms was GCC 4.8.4 provided by the Arduino Development Kit with flags `-O3 -nostdlib -fno-schedule-insns -mcpu=cpu -mthumb`, for values of `cpu` matching the processor (`cortex-m0plus/m3/m4`). For the higher-end platforms, GCC 6.3.1 provided by the operating system was used instead with common flags `-O3 -fno-schedule-insns -march=native`.

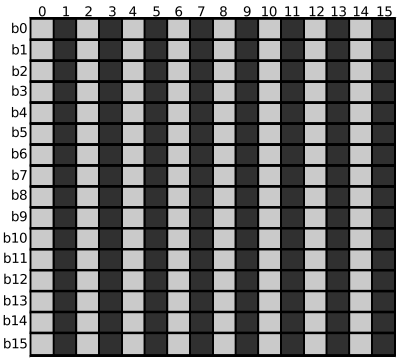
[1] `https://github.com/rafajunio/fantomas-x86`

Listing 10: Linear layer vectorized to simultaneously process 16 blocks.

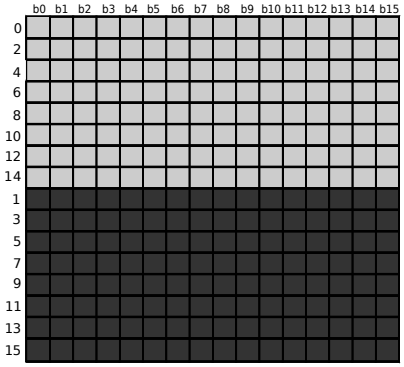
```
static inline void LLayer(v16qu *X) {
    static const v16qu tables[8] = //Store L-box in register tables
    {{0x00,0xFF,0x90,0x6F,0x37,0xC8,0xA7,0x58,0x48,0xB7,0xD8,0x27,0x7F,0x80,0xEF,0x10},
     {0x00,0xBF,0x6E,0xD1,0x41,0xFE,0x2F,0x90,0xD5,0x6A,0xBB,0x04,0x94,0x2B,0xFA,0x45},
      ...
    };
    for(int i=0; i<8; i+=2) { //Process 2 blocks in parallel
        //X[i] contains the less significant byte of the i-th block
        //X[i+8] contains the most significant byte of the i-th block
        v16qu t[4] = {X[i], X[i+8], X[i+1], X[i+9]};
        //Replace the 4 less significant bits of t[0]/t[2]
        X[i]   = __builtin_shuffle(tables[0], t[0]);
        X[i+1] = __builtin_shuffle(tables[0], t[2]);
        X[i+8] = __builtin_shuffle(tables[1], t[0]);
        X[i+9] = __builtin_shuffle(tables[1], t[2]);

        //Replace the 4 most significant bits of t[0]/t[2]
        t[0] >>= 4; t[2] >>= 4;
        X[i]   ^= __builtin_shuffle(tables[2], t[0]);
        X[i+1] ^= __builtin_shuffle(tables[2], t[2]);
        X[i+8] ^= __builtin_shuffle(tables[3], t[0]);
        X[i+9] ^= __builtin_shuffle(tables[3], t[2]);
        ...
    }
}
```

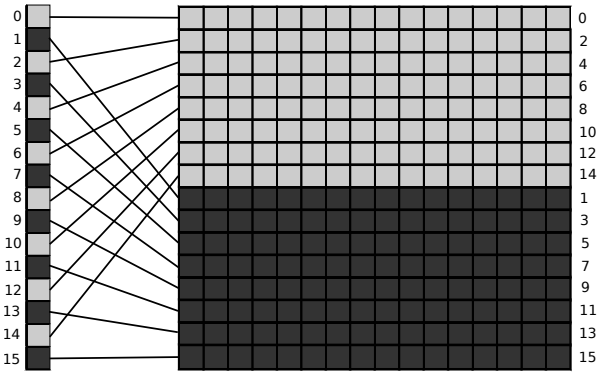
Fig. 2: Counter and key transformation for the vectorized CTR implementation. In the initial state in (a), each row represents a different counter (block $b_i$) and each column represents bytes in the same position in all counters. The final state is computed through a partial transposition of the initial state, reaching the intermediate state in (b). The key is transformed analogously in (c). An alternative figure depicting the state organization can be found in [CA16].




(a) Initial state of the counter.




(b) Final state of the counter.




(c) Key transformation.

More details about the platforms can be found below:

- **Cortex-M0+:** Arduino Zero powered by an Atmel SAMD21 ARM Cortex-M0+ 48MHz CPU. Execution time was measured through the native SysTick cycle counter.
- **Cortex-M3:** Arduino Due powered by an Atmel SAM3X8E ARM Cortex-M3 84MHz CPU. Execution time was measured by converting the output of the `micros()` function in Arduino for measuring microseconds to cycles through simple multiplication by the nominal frequency.
- **Cortex-M4:** Teensy 3.2 board containing a Cortex-M4 MK20DX256VLH7 72MHz processor. Execution time was measured through the native cycle counting register and some Assembly code.
- **Cortex-A15:** ODROID-XU4 board containing a Samsung Exynos5422 Cortex-A15 2GHz and Cortex-A7 8-core CPUs. We installed the official distribution of Arch Linux for the board, which comes equipped with GCC 6.3.1 for ARM, using the additional flags `-mfpu=neon -mcpu=cortex-a15`. Execution time was measured by enabling reading from the Cycle CouNT register (CCNT) from the Performance Monitor Unit (PMU) in user level.
- **Cortex-A53:** ODROID-C2 board containing an Amlogic ARM Cortex-A53(ARMv8) 2GHz 4-core CPUs. We installed Arch Linux with GCC 6.3.1 for ARM using flags `-mfpu=neon -mcpu=cortex-a53`. Execution time was also measured through the PMU enabled by loading a special kernel module.
- **Core i7 Ivy Bridge:** Intel Core i7-3632Q 2.20GHz CPU. GCC 6.3.1 was again used with flags `-mssse3 -msse`. The RDTSC register was used for cycle counting and Turbo Boost was disabled.
- **Core i7 Haswell:** Intel Core i7-4770 3.40GHz CPU. GCC 6.3.1 was again used with flags `-mssse3 -msse`. The RDTSC register was used for cycle counting and Turbo Boost was disabled.

Tables 1 and 2 in the next pages present results for the 32- and 64-bit portable implementations; and the NEON and SSE vectorized implementations of Fantomas. All measurements take into account the time to encrypt and decrypt using the operating modes CBC and CTR. The isochronous/constant-time implementations receive the CT abbreviation suffix. The vector implementations are intrinsically isochronous by operating over registers only. Cycle counts were computed by encrypting or decrypting the same message of length 1024 bytes a 100 times. The final result represents the average time to encrypt or decrypt a single byte using a specific implementation. Because Fantomas does not have a key schedule and encryption/decryption algorithms are very similar performance-wise, results can be easily converted to a cycles per byte (CPB) metric.

The isochronicity property of the constant time implementations was validated using the FlowTracker tool for static analysis [RPA16] and `dudect` for dynamic analysis [RBV17] in the Intel platforms. FlowTracker performs information flow analysis from function inputs marked as secret to branch instructions and memory addresses at the LLVM IR level, effectively detecting and thwarting timing attacks in compiled code. The tool `dudect` performs statistical testing of execution times. 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 Cortex-M0 to M7. However, we only report timings collected in the widely available platforms to simplify comparisons with future competing implementation efforts. Our implementations were tailored for ARM processors and enjoy the benefits of the second-operand barrel shifter.

### 5.1 Discussion

The tables contain several interesting results to be discussed. We omit cases when there is a mismatch between the word size in the implementation and the processor word, because the compiler may degrade performance substantially. This effect was very clear when compiling the 64-bit constant-time implementations on the 32-bit processors, for example.

Cache protection heuristics have a low impact of performance, with the cache-filling implementation being faster than the compact small-table one. These implementations do not have formal side-channel resistance guarantees and rely on compiler alignment and other runtime characteristics, hence they should be used only when some side-channel resistance is desired and the performance overhead of isochronicity is prohibitive.

Constant-time implementations with uniform access to memory receive a massive performance penalty. In the Cortex-M, the 32-bit CBC/CTR constant time implementation of Fantomas proved to be almost twice as compact due to the lack of precomputed tables, although more than 3 times slower than the unprotected version. Similar ratios can be found in the 32-bit Cortex-A15, but performance degradation is lower in the 64-bit Cortex-A53. If the main objective is to obtain a smaller code fingerprint and/or resistance against timing-based side-channel attacks, this implementation can however still be a good choice. Observe that Cortex-A processors and even some Cortex-M microcontrollers may have cache memory, so it is important to measure the performance impact of protecting the implementations against cache-timing leakage.

Table 1: Execution time and code size (ROM bytes) for **encryption** using Fantomas benchmarked in various Intel and ARM platforms. Figures present average cycles for encrypting a single byte (CPB) in CBC/CTR mode. There are multiple types of implementations: constant-time (CT), cache protection heuristics (small table and cache-filling) and vectorized using NEON/SSE instruction sets. For related works, we list the most efficient and compact implementations, and also implementations with a good trade-off between speed and size, as represented by the Figure of Metric (FOM) defined by FELICS [DCK+15].

| | **Cortex-M0+** | | **Cortex-M3** | | **Cortex-M4** | | **Cortex-A15** | | **Cortex-A53** | | **i7 Ivy Bridge** | | **i7 Haswell** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** |
| 32/64 (CBC) | 265.20 | 1658 | **170.46** | **1618** | 129.26 | 1618 | 49.91 | 1944 | 75.91 | 1720 | 50.96 | 1680 | 42.68 | 1680 |
| 32/64 (CTR) | 265.67 | 1808 | **171.31** | **1834** | 129.78 | 1822 | 51.22 | 2256 | 76.07 | 2008 | 50.63 | 1776 | 42.04 | 1776 |
| 32/64 (2/4-block CTR) | 205.48 | 3208 | 143.22 | 3364 | 113.50 | 3238 | 40.61 | 4108 | 73.65 | 5284 | 23.72 | 4571 | 20.42 | 4571 |
| 32/64 Compact (CBC) | 401.94 | 982 | **288.28** | **954** | 211.02 | 954 | 78.31 | 1328 | 138.83 | 1076 | 69.83 | 1130 | 61.53 | 1130 |
| 32/64 Compact (CTR) | 402.41 | 1132 | **289.07** | **1170** | 211.53 | 1158 | 78.92 | 1640 | 139.38 | 1364 | 67.06 | 1226 | 61.10 | 1226 |
| 32/64 Cache-filling (CBC) | 354.36 | 1830 | 244.11 | 1798 | 182.21 | 1798 | 59.27 | 2080 | 87.58 | 1812 | 53.01 | 1764 | 45.51 | 1764 |
| 32/64 Cache-filling (CTR) | 354.82 | 1980 | 244.95 | 2014 | 182.73 | 2002 | 59.79 | 2392 | 93.36 | 1364 | 53.19 | 1226 | 45.35 | 1226 |
| 32/64 CT (CBC) | 1132.72 | 1018 | 615.83 | 974 | 491.30 | 970 | 435.54 | 1544 | 221.85 | 1504 | 119.23 | 1853 | 103.39 | 1853 |
| 32/64 CT (CTR) | 1133.19 | 1168 | 616.68 | 1190 | 491.82 | 1174 | 441.35 | 1856 | 222.51 | 1792 | 112.82 | 1949 | 102.71 | 1949 |
| 32/64 CT (2/4-block CTR) | 1074.45 | 2348 | 566.46 | 2524 | 455.58 | 2398 | 418.92 | 3408 | 223.85 | 6012 | 79.60 | 6821 | 70.67 | 6821 |
| POPCNT (CTR) | – | – | – | – | – | – | – | – | – | – | 224.61 | 2231 | 193.10 | 2231 |
| NEON/SSE (CBC) | – | – | – | – | – | – | 63.77 | 924 | 59.98 | 680 | 55.80 | 669 | 47.97 | 669 |
| NEON/SSE (CTR) | – | – | – | – | – | – | 63.72 | 1236 | 59.68 | 968 | 49.31 | 765 | 48.14 | 765 |
| NEON/SSE (16-block CTR) | – | – | – | – | – | – | **16.05** | 6822 | 17.93 | 3868 | **5.87** | 6099 | 5.62 | 5940 |
| ***Related work*** | | | | | | | | | | | | | | |
| 32 (CBC) Fast[1] | – | – | 228.96 | 3124 | – | – | – | – | – | – | – | – | – | – |
| 32 (CTR) Fast[1] | – | – | 237.31 | 3092 | – | – | – | – | – | – | – | – | – | – |
| 32 (CBC) FOM[1] | – | – | 344.12 | 1484 | – | – | – | – | – | – | – | – | – | – |
| 32 (CTR) FOM[1] | – | – | 344.06 | 1524 | – | – | – | – | – | – | – | – | – | – |
| 32 (CBC) Compact[1] | – | – | 432.06 | 1564 | – | – | – | – | – | – | – | – | – | – |
| 32 (CTR) Compact[1] | – | – | 437.31 | 1428 | – | – | – | – | – | – | – | – | – | – |
| NEON/SSE (16-block CTR)[2] | – | – | – | – | – | – | 14.2* | – | – | – | 4.2* | – | – | – |

[1] [DCK+15] [2] [GLS+15a] * Timings provided only for reference, due to incompatible metrics or benchmarking strategies.

Table 2: Execution time and code size (ROM bytes) for **decryption** using Fantomas benchmarked in various Intel and ARM platforms. Figures present average cycles for encrypting a single byte (CPB) in CBC/CTR mode. There are multiple types of implementations: constant-time (CT), cache protection heuristics (small table and cache-filling) and vectorized using NEON/SSE instruction sets. For related works, we list the most efficient and compact implementations, and also implementations with a good trade-off between speed and size, as represented by the Figure of Metric (FOM) defined by FELICS [DCK+15].

| | **Cortex-M0+** | | **Cortex-M3** | | **Cortex-M4** | | **Cortex-A15** | | **Cortex-A53** | | **i7 Ivy Bridge** | | **i7 Haswell** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** | **Cycles per byte (CPB)** | **Code size (Bytes)** |
| 32/64 (CBC) | 255.64 | 1634 | **182.18** | **1598** | 144.08 | 1598 | 51.73 | 1944 | 85.96 | 1752 | 51.02 | 1662 | 46.51 | 1662 |
| 32/64 (CTR) | 265.67 | 1524 | **171.31** | **1492** | 129.78 | 1492 | 51.22 | 1724 | 76.07 | 2008 | 50.63 | 1776 | 42.04 | 1776 |
| 32/64 (2/4-block CTR) | 205.48 | 3208 | 143.22 | 3364 | 113.50 | 3238 | 40.61 | 4108 | 73.65 | 5284 | 23.72 | 4571 | 20.42 | 4571 |
| 32/64 Compact (CBC) | 373.41 | 970 | **286.24** | **938** | 216.69 | 938 | 83.55 | 1296 | 122.49 | 1092 | 68.23 | 1110 | 64.42 | 1110 |
| 32/64 Compact (CTR) | 402.41 | 1132 | **289.07** | **1170** | 211.53 | 1158 | 78.92 | 1640 | 139.38 | 1364 | 67.06 | 1226 | 61.10 | 1226 |
| 32/64 Cache-filling (CBC) | 303.41 | 1714 | 247.25 | 1726 | 181.99 | 1726 | 61.43 | 2076 | 95.50 | 1832 | 49.07 | 1754 | 45.68 | 1754 |
| 32/64 Cache-filling (CTR) | 354.82 | 1980 | 244.95 | 2014 | 182.73 | 2002 | 59.79 | 2392 | 93.36 | 2100 | 53.19 | 1860 | 45.35 | 1860 |
| 32/64 CT (CBC) | 1127.86 | 942 | 610.59 | 898 | 486.39 | 898 | 442.38 | 1528 | 230.92 | 1500 | 136.66 | 1849 | 121.26 | 1849 |
| 32/64 CT (CTR) | 1133.19 | 1168 | 616.68 | 1190 | 491.82 | 1174 | 441.35 | 1856 | 222.51 | 1792 | 112.82 | 1949 | 102.71 | 1949 |
| 32/64 CT (2/4-block CTR) | 1074.45 | 2348 | 566.46 | 2524 | 455.58 | 2398 | 418.92 | 3408 | 223.85 | 6012 | 79.60 | 6821 | 70.67 | 6821 |
| POPCNT (CTR) | – | – | – | – | – | – | – | – | – | – | 224.61 | 2231 | 193.10 | 2231 |
| NEON/SSE (CBC) | – | – | – | – | – | – | 73.19 | 908 | 63.20 | 684 | 53.34 | 663 | 53.32 | 663 |
| NEON/SSE (CTR) | – | – | – | – | – | – | 63.72 | 1236 | 59.68 | 968 | 49.31 | 765 | 48.14 | 765 |
| NEON/SSE (16-block CTR) | – | – | – | – | – | – | **16.05** | 6822 | 17.93 | 3868 | **5.87** | 6099 | 5.62 | 5940 |
| ***Related work*** | | | | | | | | | | | | | | |
| 32 (CBC) Fast[1] | – | – | 189.63 | 4316 | – | – | – | – | – | – | – | – | – | – |
| 32 (CTR) Fast[1] | – | – | 237.31 | 3092 | – | – | – | – | – | – | – | – | – | – |
| 32 (CBC) FOM[1] | – | – | 346.24 | 2148 | – | – | – | – | – | – | – | – | – | – |
| 32 (CTR) FOM[1] | – | – | 344.06 | 1524 | – | – | – | – | – | – | – | – | – | – |
| 32 (CBC) Compact[1] | – | – | 487.02 | 2138 | – | – | – | – | – | – | – | – | – | – |
| 32 (CTR) Compact[1] | – | – | 437.31 | 1428 | – | – | – | – | – | – | – | – | – | – |
| NEON/SSE (16-block CTR)[2] | – | – | – | – | – | – | 14.2* | – | – | – | 4.2* | – | – | – |

[1] [DCK+15] [2] [GLS+15a] * Timings provided only for reference, due to incompatible metrics or benchmarking strategies.

It was surprising that the 64-bit implementation in the Cortex-A53 was slower than the 32-bit implementation in the Cortex-A15, but the Cortex-A53 is at the low-end of the 64-bit ARM processors, so better performance for the 64-bit implementation might be expected from higher-end processors.

Code sizes generally grow from the Cortex-M0+ to the Ivy Bridge. The code size for the 16-block NEON (CTR) implementation in the Cortex-A53 was also surprising, producing almost twice more compact binaries than the same NEON version in the Cortex-A15 and SSE version in the Core i7. There is a clear space-time trade-off in the multi-block CTR implementations. They are the largest implementations in terms of code size, but also almost always the fastest in a given platform, specially in those supporting vector instructions.

For the masked implementations of Fantomas, cycle counts for encrypting one block with different numbers of shares are presented in Figure 3, where a clear quadratic trend for the performance degradation can be observed, as expected [GLSV14]. The figure also illustrates the impact of random number generation, as generating random bytes for the masked `AND` operations in the S-boxes can consume 97% of the execution time. Three versions were actually implemented: table-based L-box, isochronous and NEON. All versions can be seen in Figure 4 and Table 3. Table 3 presents the timings required to encrypt a single block with $d$ shares using all three versions without random number generation. Values for $d = 1$ represent the time to encrypt a single block in Fantomas without masking.

Fig. 3: Cycle counts for encrypting one 128-bit block with the masked implementation of Fantomas in the Cortex-A15 platform as a function of the number of shares. The black dots take into account the time to generate random numbers with the Hash_DRBG using SHA256, while the red dots disregard random number generation. Both red and black points use the table-based L-Box.

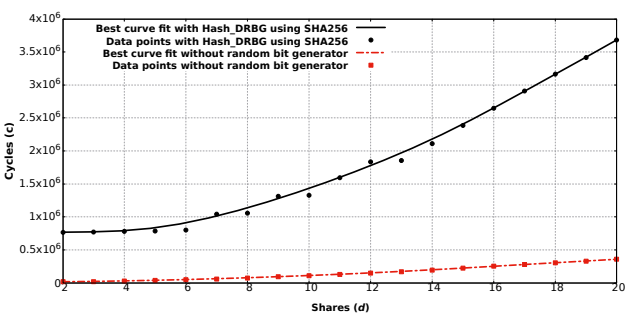



Fig. 4: Cycle counts for encrypting one 128-bit block with the masked implementation of Fantomas in the Cortex-A15 platform as a function of the number of shares. The black dots refer to the table-based L-box, red dots to the isochronous and the blue dots to the NEON version.

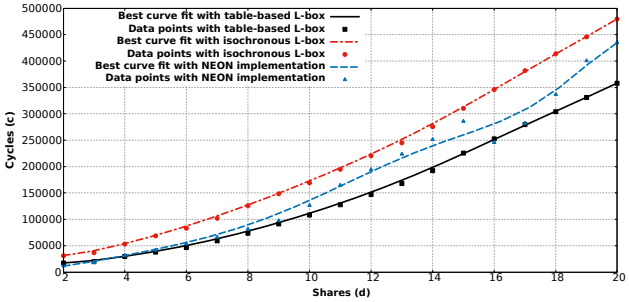



Table 3: Cycle counts for encrypting one 128-bit block with the masked implementation of Fantomas using masked 32 bits, 32-bit isochronous (CT) and NEON in the Cortex-A15 platform.

| **Shares** ($d$) | **32 bits (cycles)** | **32 bits CT (cycles)** | **NEON (cycles)** |
|---|---|---|---|
| **1** | 789 | 6854 | 990 |
| **2** | 17285 | 31382 | 11411 |
| **3** | 19534 | 36743 | 18727 |
| **4** | 29543 | 53191 | 31844 |
| **5** | 38074 | 68424 | 41754 |
| **6** | 46664 | 82970 | 53925 |
| **7** | 59015 | 101782 | 66189 |
| **8** | 73739 | 125908 | 81523 |
| **9** | 91238 | 148323 | 97799 |
| **10** | 108276 | 169223 | 126548 |

## 5.2 Comparison with related work

There are two main related works that established the previous state of the art in the context of this paper. The most recent is the massive implementation effort from the FELICS framework [DCK+15] to compare lightweight block ciphers performance-wise in representative 8/16/32-bit platforms. The project website[2] also contains results for some stream ciphers and block ciphers underlying MAC constructions. The target 32-bit

[2] `https://www.cryptolux.org/index.php/FELICS`

platform considered in their work is the same Cortex-M3 present in the Arduino Due and two scenarios are taken into consideration. Scenario 1 considers consecutive encryption and decryption of 128 bytes in CBC mode, simulating a communication protocol. In the website, the best implementation according to their Figure of Merit (FOM) takes for encryption 44,047 cycles using 1484 bytes of ROM (or 344.12 CPB in Table 1) and for decryption 44,319 cycles using 2148 bytes of ROM (or 346.24 CPB in Table 2). Even in comparison with the fastest and most compact implementation from the website our implementation is still competitive, being 25.6% to 60.5% more efficient in encryption and 3.9% to 62.6% in decryption than their implementations, and competitive in terms of code size with their more compact implementation. These speedups decreased in comparison to our previous work [CA16] due to performance improvements from FELICS. In Scenario 2, FELICS reports a range of figures for unprotected Fantomas when encrypting 128 bits in CTR mode, ranging from most compact implementation to best execution time. The most compact takes 6,997 cycles and 1428 bytes of ROM (437.31 CPB), the most efficient takes 3,797 cycles and 3,092 bytes of ROM (237.31 CPB) and a good trade-off is found at 5,505 cycles and 1524 bytes of code size (344.06 CPB). After the proper conversions, our implementation improves these figures by 60.8%, 27.8% and 50.2%, respectively, by spending only 1834 bytes of ROM.

Our compact small-table implementation takes 288.3 CPB for encryption with only 954 bytes of ROM and 286.2 CPB for decryption with only 938 bytes of ROM, both in Scenario 1; and 289.07 CPB with 1170 bytes of ROM in Scenario 2. The compact version is 33.3% and 41.2% faster in Scenario 1 and 33.9% in Scenario 2 than the compact version of FELICS, respectively, using only 1170 bytes in CTR mode. An even more compact version with only 870 bytes of ROM in CTR mode was also implemented but the speed degrades to 426.46 CPB, although still faster than FELICS. The same version is competitive in CBC mode with only 1262 bytes of ROM adding encryption and decryption. As a reference point, FELICS reports much higher latencies for standardized block ciphers such as AES under different operating modes (30,613 cycles or 239.16 CPB for encrypting and 42,114 cyles or 329.017 CPB for decrypting in CBC mode in the Cortex-M3, for example).

The second related work is the presentation for the SCREAMv3 candidate in the CAESAR competition [GLS+15a]. In the slides, numbers for a 16-block vector implementation of Fantomas are also reported. We could not reproduce the numbers presented in the table due to unavailability of the Fantomas code, and benchmarking the publicly-available SCREAMv3 code gave rather different results. In private contact with the authors, we discovered that their benchmarking code takes the outputs of the `gettimeofday()` function for time measurement, a less precise approach than using cycle counts measured directly. Additionally, it is not clear if their numbers were taken in a machine with Turbo Boost enabled, as it is well known to distort benchmarking data [BL16]. Due to the different benchmarking strategies, we only report the numbers for reference, without attempting an explicit comparison.

### 5.3 Comparison with Fantomas*

After the LS-Design block cipher Robin was successfully attacked through invariant-subspace attacks [LMR15], the algorithm was tweaked to include larger tables of round constants spanning the entire internal state. This modification was called Robin*. Although Fantomas was not affected by the same attack, the same countermeasure was applied to the algorithm in a follow-up work [JSV17], leading to the version called Fantomas*.

We have implemented and benchmarked Fantomas* in some representatives target architectures, more specifically the Cortex-M4 and Cortex-A15, and observed small performance penalties due to the extra additions of round constants. The largest performance difference was 10.3% in the execution of insecure table-based implementations, since addition of round constants responds to a higher part of overall performance. Constant-time versions suffered penalties of at most 2%. The least affected versions were the vectorized NEON implementations, in which performance became only 0.1% worse.

We could not compare our masked implementation with timings from [JS17] in the Cortex-M4, because their implementation employs very different techniques: the masking technique is novel and more efficient, and implemented in Assembly at a much higher order. There are also differences in the target platform configuration and how random number generation is implemented. From the point of view of security, it is worth noting that our portable code written in C may suffer from further order reduction exacerbated by the compiler [BGG+14].

## 6 Instruction Set Extension

The previous sections studied software-based countermeasures for side-channel resistance. This section adopts a different approach and presents a hardware extension built to enable more secure and efficient implementations of Fantomas. Although our software solu-

tions could already be sufficiently effective on mitigating the side-channel vulnerabilities, an extended hardware support allows the implementations to achieve the same security property while reducing the performance penalty.

For this experiment, we built a system using Intel Platform Designer [Int17] (formerly Altera Qsys) and instantiated an Altera NIOS II processor [Alt16], along with other Altera IP components and our hardware extension. Table 4 lists all the instantiated components and their configurations.

Our hardware modifications were guided by two goals: to improve the performance of the online L-box evaluation and to mitigate the side-channel vulnerability in the L-box table lookups. This way, two different secure versions were produced to explore hardware features and trade-offs. We achieve the first goal by introducing a new specialized instruction for parity check, which enabled a great performance improvement in the online parity calculation. The second goal, in turn, was achieved by simply activating NIOS' cache bypass mechanism, which allowed the L-box table accesses to be performed in constant time.

Table 4: System components and configurations.

| Component | Configuration |
|---|---|
| Clock Source | Known frequency - 50 MHz |
| NIOS II Processor - Gen 2 | NIOS II/f<br>Cache configured according to the experiment.<br>All other configurations to default. |
| JTAG UART | Default |
| On-Chip Memory | RAM - Size: 159.744 bytes |
| Performance Counter | 4 simultaneously-measured sections |
| System ID | Default |
| Interval Timer | Default |
| Custom Instruction Type | Extended (Combinational) |

### 6.1 Cache bypass mechanism

The NIOS processor has a built-in mechanism for cache bypass that can be activated through its configuration interface. The mechanism works by checking the value of the most significant bit of a memory address before looking up to it in the cache: If the bit is set, the cache is ignored and a direct access is performed to the main memory. Otherwise, the ordinary cache look-up process continues.

It should be noted that, despite being a 32-bit processor, NIOS is capable of addressing only 31-bit addresses, once the most-significant bit is reserved for the cache bypass feature. Listing 11 shows the macro used for setting the most significant bit.

### 6.2 Parity check instruction

NIOS also has an interface for extending its instruction set. It can communicate with the processor using up to 15 signals defined by standardized templates. For the parity instruction, we used the simplest one, designed for combinational instructions. It offers two 32-bit inputs as instruction operands and one 32-bit output as instruction result. We also added the `n` input, which receives a 3-bit value representing an opcode offset. The implementation logic is similar to the one presented in Listing 7. We synthesized our system to a Cyclone V SoC 5CSEMA4U23C6N FPGA device, where the instruction took 25 Adaptive Look-Up Tables (ALUTs). For comparison, the entire system we built took 2734 ALUTs.

The instruction behaves as follows:

1. It receives two operands, `dataa` and `datab` and calculates a logical `AND` between them.
2. Depending on the value of the opcode offset $\texttt{n} \in 0, 1, 2, 3$, the instructions performs $2^{4-n}$ parity check computations on $2^{n+1}$ chunks; otherwise, the instruction performs one parity check calculation on the entire register.
3. The result is presented in the least-significant bit of each chunk.

The first step was conveniently chosen to help the implementation of Fantomas, but it does not affect the instruction generality. The parity chunk size, implemented using the opcode offset, is also not necessary for Fantomas, since we only use $\texttt{n} = 3$. Nonetheless, we have decided to implement this feature once it brings more flexibility to the instruction at almost zero hardware cost.

Listing 12 defines the macro `ALT_CI_PARITY_0` used in the C code for inserting the parity check custom instruction. The macro receives the operands `A` and `B`, the opcode offset `n`, and returns the result. A built-in compiler function is used to perform the insertion of the instruction.

### 6.3 Experiments using the NIOS II processor

Using the system described in Table 4, we executed the following experiments:

– **Baseline:** Non-modified versions for performance comparison.

- **Parity Instruction:** Implementations using our specialized custom instruction to perform the online parity check calculation.
- **Cache Bypass:** Implementations using the cache bypass mechanism for the L-box access.
- **No-Cache:** Implementations executed with the processor's cache disabled.

Table 5 shows the results of each experiment. The first column indicates the versions in which the L-box is either evaluated *Online* or when it is queried from a *Table*. For the *Baseline* experiment, *Table* versions are non-constant time. Analyzing the results, it can be noted that the *Online* versions had a 62% gain on average when using the specialized instruction rather than the C implementation for the parity calculation. Considering the *Table* versions, we had a 25% slowdown when using the cache bypass mechanism to mitigate the side-channel vulnerability. From these results, the best approach to take still seems to be using precomputed L-boxes. The *Table* versions are 47% faster than the *Online* ones when using the cache bypass mechanism and 18% faster with the entire cache disabled.

It should be mentioned, though, that NIOS has a relatively small cost for memory accesses due to its low frequency. Thus, the conclusions obtained in this section could be different for other architectures.

Listing 11: Cache bypass macro.

```
#define DONT_CACHE(A)                                  \
  uint16_t *A##_t = A;                                 \
  uint16_t *A = (uint16_t *)((uint32_t)A##_t \
  | (uint32_t) ALT_CPU_DCACHE_BYPASS_MASK);
```

Listing 12: Custom instruction macro.

```
#define ALT_CI_PARITY_0_N                    0x0
#define ALT_CI_PARITY_0_N_MASK      ((1<<3)-1)
#define ALT_CI_PARITY_0(n,A,B)                 \
  __builtin_custom_inii(ALT_CI_PARITY_0_N + \
  (n&ALT_CI_PARITY_0_N_MASK),(A),(B))
```

## 7 Conclusion

We presented several serial and vectorized software implementations of the Fantomas block cipher, producing more efficient and compact implementations in the ARM and Intel target platforms. Four approaches for side-channel resistance were implemented: constant time, cache protection heuristics, masking and hardware extensions. The constant time approach for implementing the L-box is of independent interest, as it can also be easily extended to other LS-Design ciphers. A simple instruction set extension for parity computation reduced the performance penalty of the constant time countermeasures by more than half. The cache protection heuristics have a lower performance impact, but lose security guarantees compared to the isochronous implementation. Masking illustrates the computational cost of powerful side-channel countermeasures. Even if Fantomas was conceived to be easily masked in a protected implementation, the performance penalty can be as high as a factor of 27x with 10 shares, when compared to a constant time implementation. We have also observe that efficient random number generation is an important research target to enable masked software implementations to perform well in a realistic setting.

As future work, we leave the task of evaluating what countermeasures are more effective against different kinds of side-channel attacks.

Acknowledgements

The first and last authors acknowledge financial support and access to the ARM MPS2 board by LG Electronics Inc. during the development of this research, under the project titled "*Efficient and Secure Cryptography for IoT*". The second and third authors acknowledge financial support from Intel and FAPESP under the project "*Secure Execution of Cryptographic Algorithms*", grant 14/50704-7. We thank the anonymous reviewers for their comments.

## References

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[Alt16] Altera. Nios ii processor reference handbook, 2016.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

Table 5: Execution time for Fantomas benchmarked in NIOS II

| Version | | Execution Time (cycles per Byte) | | | |
|---|---|---|---|---|---|
| L-box | Name | Baseline | Parity Instruction | Cache Bypass | No-Cache |
| Online | CBC Encryption | 875.69 | 335.60 | - | - |
| | CBC Decryption | 880.23 | 331.99 | - | - |
| | CTR | 874.53 | 340.58 | - | - |
| | 2-block CTR | 846.51 | 303.67 | - | - |
| Table | CBC Encryption | 146.80 | - | 183.57 | 263.41 |
| | CBC Decryption | 147.96 | - | 185.52 | 285.46 |
| | CTR | 141.68 | - | 184.33 | 264.61 |
| | 2-block CTR | 114.36 | - | 138.56 | 259.85 |
| | CBC Compact Encryption | 216.12 | - | 316.68 | 437.91 |
| | CBC Compact Decryption | 213.27 | - | 309.35 | 418.06 |
| | CTR Compact | 221.60 | - | 322.22 | 443.69 |


[Ber04] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *CARDIS*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.

[BK12] Elaine Barker and John Kelsey. NIST SP 800-90A – Recommendation for Random Number Generation Using Deterministic Random Bit Generators, 2012.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BL16] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`, 2016.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.

[CA16] Rafael J. Cruz and Diego F. Aranha. Efficient Software Implementations of Fantomas. In *16th Brazilian Symposium on Information and Computer Systems Security (SBSeg 2016)*, pages 212–225, 2016.

[CDL15] Anne Canteaut, Sébastien Duval, and Gaëtan Leurent. Construction of lightweight s-boxes using feistel and MISTY structures. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*, pages 373–393. Springer, 2015.

[DCK+15] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of lightweight block ciphers for the internet of things. Cryptology ePrint Archive, Report 2015/209, 2015. `http://eprint.iacr.org/`.

[DPU+16] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design strategies for ARX with provable bounds: Sparx and LAX. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 484–513, 2016.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[Fog16] A. Fog. Instruction tables: List of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/instruction_tables.pdf`, version published on 08 Oct 2018., 2016.

[GLS+15a] Vincent Grosso, Gaëtan Laurent, François-Xavier Standaert, Kerem Varici, François Durvaux, Lubos Gaspar, and Stéphanie Kerckhof. CAESAR candidate SCREAM Side-Channel Resistant Authenticated Encryption with Masking. `http://2014.diac.cr.yp.to/slides/leurent-scream.pdf`, 2015.

[GLS+15b] Vincent Grosso, Gaëtan Laurent, François-Xavier Standaert, Kerem Varici, François Durvaux, Lubos Gaspar, and Stéphanie Kerckhof. SCREAM Side-Channel Resistant Authenticated Encryption with Masking. `https://competitions.cr.yp.to/round2/screamv3.pdf`, 2015.

[GLSV14] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. Ls-designs: Bitslice encryption for efficient masked software implementations. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2014.

[Int17] Intel. Quartus prime standard edition handbook volume 1 - design and synthesis, 2017.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[JS17] Anthony Journault and François-Xavier Standaert. Very high order masking: Efficient implementation and security evaluation. In *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 623–643. Springer, 2017.

[JSV17] Anthony Journault, François-Xavier Standaert, and Kerem Varici. Improving the security and efficiency of block ciphers based on LS-designs. *Des. Codes Cryptography*, 82(1-2):495–509, 2017.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[LMR15] Gregor Leander, Brice Minaud, and Sondre Rønjom. A generic approach to invariant subspace attacks: Cryptanalysis of robin, iscream and zorro. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 254–283. Springer, 2015.

[PRC12] Gilles Piret, Thomas Roche, and Claude Carlet. PICARO - A block cipher allowing efficient higher-order side-channel resistance. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, volume 7341 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2012.

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *DATE*, pages 1697–1702. IEEE, 2017.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

[RPA16] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 110–120. ACM, 2016.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732. USENIX Association, 2014.