Efficient Software Implementations of Fantomas

Rafael J. Cruz[1], **Diego F. Aranha**[1]

[1]Laboratory of Security and Applied Cryptography (LASCA)

Institute of Computing (IC) – University of Campinas (Unicamp)

Av. Albert Einstein, 1251 – Campinas/SP – Brazil

{raju,dfaranha}@lasca.ic.unicamp.br

Abstract. *We present a series of software implementations of the Fantomas block cipher in resource-constrained ARM devices like the Cortex-M3 and Cortex-M4; and more powerful processors such as the ARM Cortex-A15 and modern Intel platforms. Our implementations span a broad range of characteristics: 32-bit and 64-bit versions, side-channel resistant and vectorized code for NEON and SSE instructions. Our implementations of the algorithm improve the state of the art substantially, both in terms of efficiency or compactness, by making use of novel algorithmic techniques and features specific to the target platform. In particular, our unprotected 32-bit implementation achieves speedups from 35% to 66% in the ARM Cortex-M architecture, while consuming considerably less code size. The vectorized implementations improve performance over the state of the art by 40% in the ARM Cortex-A15 and 50% in the Core i7 Ivy Bridge, setting new speed records for the implementation of the block cipher.*

1. Introduction

Lightweight cryptography for embedded systems has been a very active field of research in the last years, and it recently gained renewed interest with the emergence of the Internet of Things. Cryptographic primitives can indeed mitigate or even solve several problems faced by connected devices collecting and exchanging sensitive information through an open network. Many innovative encryption algorithms were proposed to maximize performance in resource-constrained devices and to provide lighter alternatives to AES [Daemen and Rijmen 2002], without compromising security. Some remarkable examples are the NSA-designed SPECK and SIMON block cipher families [Beaulieu et al. 2013], the PRINCE block cipher [Borghoff et al. 2012], and more recently the Low-power Encryption Algorithm (LEA) [Hong et al. 2014]. These lightweight designs follow multiple constructions, such as Feistel, Substitution-Permutation and ARX networks, posing distinct trade-offs in terms of efficiency, compactness and resistance against different attacks. While these algorithms are still considered secure according to the latest cryptanalytic results, their corresponding implementations may be susceptible to attacks based on information leakage.

Side-channel analysis is a growing and important issue for security in cryptography, specially in embedded devices. These attacks are based on information leaked during computation through side channels such as execution time, energy consumption, acoustic and electromagnetic emanations. When successful, they help the adversary to identify and recover secret data from observations captured from implementations of cryptography, 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 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 [Aciiçmez et al. 2007] or Simple Power Attacks (SPA); or require traces from many consecutive observations, as in the case of Differential Power Attacks (DPA) [Kocher et al. 1999]. Resistance to side-channel attacks has been considered as an additional security requirement for low-cost ciphers, because the lightweight devices implementing them may be physically accessible to the attacker. Algorithms with side-channel resistance guarantees embedded in the construction itself have been thus favored in the scientific literature, bringing attention to ciphers like PICARO [Piret et al. 2012] and Fantomas [Grosso et al. 2015c].

The LS-Design paradigm [Grosso et al. 2015c] was created with side-channel resistance in mind, because it allows the designer to construct lightweight algorithms friendly to efficient 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 techniques with controlled overhead. Masking schemes were initially proposed in 2003 [Ishai et al. 2003] in the context of protecting circuits against probing, but it has been later extended to much more complex operations, even achieving provable security guarantees [Rivain and Prouff 2010]. 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 perturbation techniques to introduce external random noise in the encryption or decryption processes, acting as countermeasure against several types of side-channel attacks.

One of the first and most famous instances of the LS-Design construction is the Fantomas block cipher. This work presents several efficient, compact, portable and secure (in the sense of side-channel resistance) implementations of Fantomas. In terms of performance, a number of optimizations are described to save execution time or code size, several of them easily adaptable to other LS-Designs, such as the CAESAR candidate SCREAMv3 [Grosso et al. 2015b]. In terms of security, constant-time and masked implementations are discussed. The constant-time implementation protects execution against timing attacks [Kocher 1996] and avoids precomputed tables vulnerable against cache latency attacks [Bernstein 2004, Bonneau and Mironov 2006]; and the masked implementation illustrates several current challenges of the research field. The constant-time implementation was validated using the FlowTracker static analysis tool [Silva et al. 2016].

This paper is organized as follows. Section 2 introduces the masking implementation strategy, LS-Designs and the Fantomas block cipher. Section 3 discusses multiple implementations of the algorithm, targeting different platforms. Section 4 presents experimental results and Section 5 concludes the paper.

2. Preliminaries

In this section, we introduce the concept of masking for protecting implementations against side-channel attacks and describe the LS-design construction instantiated by the Fantomas block cipher.

2.1. Masking Scheme

Masking is one of the most investigated countermeasures against side-channel cryptanalysis. In the context of block ciphers, masking aims to protect sensitive data, such as plaintext during encryption, or the ciphertext during decryption. Because information computed in these processes will be later transformed into the algorithm outputs, all intermediary states must be protected at all time. The masked state of m with $d+1$ *shared secrets* is given by $m = \bigoplus_{i=0}^{d} m_i = m_0 \oplus m_1 \oplus \ldots \oplus m_d$, 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 on ciphers employing operations in finite field $\mathbb{F}_2$:

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

$$L(m) \equiv L(m_0 \oplus m_1 \oplus \ldots \oplus m_d) \equiv L(m_0) \oplus L(m_1) \oplus \ldots \oplus L(m_d)$$

2. A `NOT` operation over a masked secret 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 as in Algorithm 1.

Algorithm 1 Non linear operation `AND` performed on two masked secrets a and b

```
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 $\oplus_{i=0}^{d} c_i = a \wedge b$
for $i$ from $0$ to $d$ do
    $r_{i,i} \leftarrow 0$;
    for $j$ from $i+1$ to $d$ do
        $r_{i,j} \leftarrow random()$;
        $r_{j,i} \leftarrow (r_{i,j} \oplus (a_i \wedge b_j)) \oplus (a_j \wedge b_i)$;
    end for
end for
for $i$ from $0$ to $d$ do
    $c_i \leftarrow a_i \wedge b_i$;
    for $j$ from $0$ to $d$ do
        $c_i \leftarrow c_i \oplus r_{i,j}$;
    end for
end for
```

These observations allow any algorithm employing binary field arithmetic to be implemented in a masked way. An important challenge in masked implementations can be seen in line 4 of the algorithm, in the form of random number generation. By considering that every share a_i is a *unity*, every masked AND requires $\frac{(d+1)^2-(d+1)}{2}$ unities of random data and additional space of $(d+1)^2$ to store a matrix containing all possible combinations of shares.

2.2. LS-Designs and Fantomas

LS-Designs were conceived to address side-channel threats, by combining the advantages of bitslicing-capable ciphers with easy support to regular and masked software implementations. Algorithm 2 presents a generic specification for an LS-Design, illustrating its simplicity and regularity. Instances of a 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 round constants $C(r)$. In the original LS-Design paper, two ciphers were instantiated and analyzed: Robin, a faster involutive instance that later succumbed to invariant subspace attacks [Leander et al. 2015]; and the non-involutive candidate Fantomas.

Algorithm 2 LS-Design construction encrypting plaintext P with key K.

```
$x \leftarrow P \oplus K$                         ▷ $x$ represents an $s \times l$-bit matrix
for $0 \leq r < N_r$ do
    for $0 \leq i < l$ do                         ▷ S-box layer
        $x[i, \star] = S[x[i, \star]]$
    end for
    for $0 \leq j < s$ do                         ▷ L-box layer
        $x[\star, j] = L[x[\star, j]]$
    end for
    $x \leftarrow x \oplus K \oplus C(r)$         ▷ Key and round constant addition
end for
return x
```

Fantomas employs the 3/5-bit S-boxes from the 3-round MISTY cipher [Canteaut et al. 2016], as presented in detail on Algorithm 3. An important consideration taken by the original authors of the cipher is the number of AND operations in the choice of S-boxes. As discussed in Section 2.1, masked implementations of the algorithm must rely on Algorithm 1 when computing ANDs. For security of the masking countermeasure, a lower bound on the number of ANDs is the size of the S-boxes. Because Fantomas employs S-boxes of 8-bit granularity, the S-boxes must contain at least 8 AND operations to be appropriate for masking. There is some security margin in this design decision because Fantomas employs 11 AND operations between elements of the cipher state. The L-box is presented in Figure 1 and its computation can be seen as a vector-matrix product in $\mathbb{F}_2$, as illustrated in the picture.

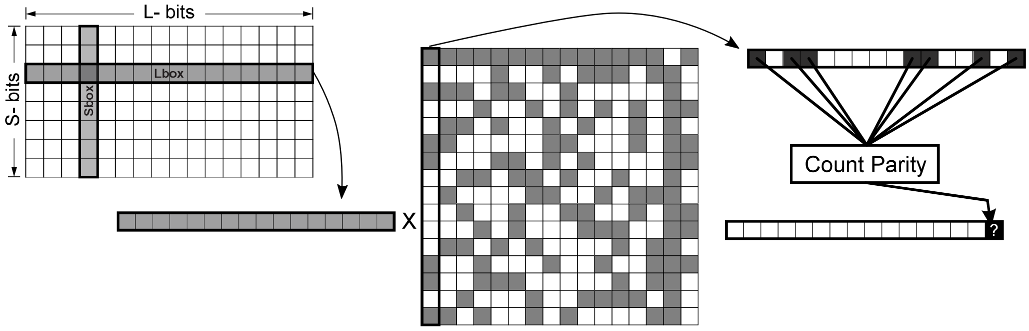


Figure 1. Linear layer of Fantomas. The L-box matrix has gray cells for `1` bits and white cells for `0` bits.

Algorithm 3 MISTY 3/5 bits S-boxes on 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. Implementation

In this section, we present the multiple implementations of the Fantomas block cipher performed by this work. We discuss portable implementations for 32-bit and 64-bit processors, mostly targeting ARM platforms, and additional code vectorized for SSE/NEON instructions. Strategies for masked implementation are discussed later, before experimental results are presented.

3.1. 32-bit implementation

We have implemented two 32-bit variants of the cipher: a constant-time version protected against timings attacks and an unprotected one. Both versions require 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. Following the C99 standard, we prevented breaking strict aliasing point rules by representing the internal state as a union combining pointers to the data types as in Figure 2.

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

Figure 2. Union to respect the strict aliasing rule: two different pointers cannot reference the same memory area.

The implementations still take aligned byte vectors as input and conveniently converts them to 32-bit pointers when needed. The S-boxes must then be computed using the union structure. Some operations over 16-bit chunks could be combined in 32-bit operations, but this was avoided to prevent unaligned loads and stores. Their bitsliced structure already provides the constant time property due to bitslicing, so no additional countermeasures were needed for secure implementation of the substitution layer.

The diffusion layer is performance-critical and presents more obstacles to side-channel resistance, since it is implemented through table lookups on the L-box. The unprotected version employs two 256-position half-word precomputed tables, while the protected version implements the operation online by performing a vector-matrix binary multiplication, where two 16-bit words are processed at the same time. A small code portion illustrating the unprotected L-box can be found in Figure 3, where `state` stores the 128-bit state, `LBoxH` transforms the 8 most significant bits and `LboxL` transforms the 8 less significant bits for all $j \in \{0, 1, 2, 3\}$. Observe that the table lookups are vulnerable to adversarial influence over the memory hierarchy in processors equipped with cache memory [Bernstein 2004, Bonneau and Mironov 2006].

```
/* Unprotected L-box version */
state[j].u16[0] = LBoxH[state[j].u16[0]>>8] ^
                  LBoxL[state[j].u16[0] & 0xff];
state[j].u16[1] = LBoxH[state[j].u16[1]>>8] ^
                  LBoxL[state[j].u16[1] & 0xff];
```

Figure 3. Unprotected L-Box using the internal states of the union. L-BoxH contains the higher 8-bit of the linear transformation and L-BoxL other less significant 8-bit.

The protected implementation is a little more involved. The code portion in Figure 4 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 `s`-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.

The key addition of Fantomas works by accumulating the key in the internal state using 32-bit XOR operations, as in Figure 5.

```
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;
}
```

Figure 4. Multiplying 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 $(x \cdot L)_s = (x_a \cdot y_s, x_b \cdot y_s)$.

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

Figure 5. Key Addition of Fantomas using the 32-bit state of the union.

3.2. 64-bit implementation

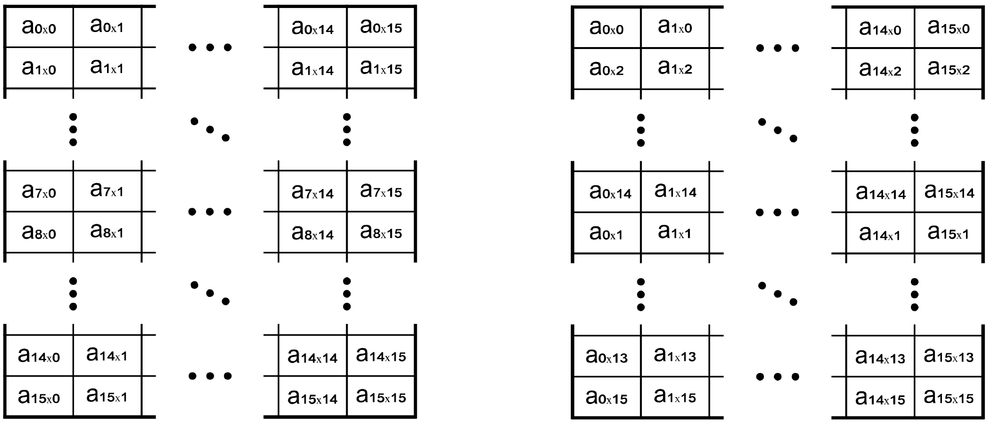
Two variants of the cipher were also implemented for 64-bit architectures. A modified union structure combines 16-bit and 64-bit words. The S-boxes must again be implemented over the union without breaking alignment and causing performance penalties. The unprotected L-box follows the same structure as the corresponding 32-bit implementation. Function `ProdLBox` was transformed to operate over 64 bits with simple modifications to the input and output types and a duplicated bit mask `0x0001000100010001` in the last operation, allowing computation of 4 simultaneous evaluations of the L-box. We also implemented a 64-bit version using the `POPCNT` instruction that proved much less efficient.

3.3. Vectorized implementation

We first discuss what structures inside a typical LS-Design are friendly to vectorization, before the vector implementations for ARM (using NEON instructions) or Intel platforms (equipped with SSE instructions) are described. 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 chunk 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 high 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 (such as Intel `PSHUFB`) for table lookups inside registers. 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. 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. Because the Fantomas block size has 128 bits (or 16 bytes), the best choice for minimum number of blocks is 16. When operating over 16 blocks simultaneously, 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.

Observe that picking a certain number of simultaneous blocks to operate has impact on the choice of block cipher operating mode, since parallelism must be supported in both encryption and decryption. The traditional CBC mode of operation imposes a serialization of encryption, thus vectorized implementations should stick to the CTR mode of operation, where only the counters are encrypted/decrypted and later added to the plaintext/ciphertext, respectively. We also implemented a single-block vectorized Fantomas for benchmarking the CBC mode of operation, using the same substitution layer described in Algorithm 3.

The L-box is a linear transformation, thus the 16 bits can be broken in smaller pieces. Hence the L-box in Figure 1 can be split in 4-bit chunks and the table reduced to 4 tables of 16 positions storing 16-bit values. To make use of the shuffle instructions mapping 4-bit sets 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. 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. 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 6a, and the state must be transposed and stored as in Figure 6b below.



(a) Initial state of the counter **(b) Final state of the counter**

Figure 6. Counter transformation for the vectorized CTR implementation.

The organization in Figure 6 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 6b and the most significant bytes in the remaining 8 blocks in the same Figure. The SSE versions of Fantomas are publicly available for independent benchmarking and reproducibility[1].

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

3.4. Masked implementation

The masked implementation needs only large modifications in the S-boxes, because every operation computed in Algorithm 3 must now be replaced by the operations specified in Section 2.1. Countermeasures are still needed for encryption and decryption, because the linear layer comes immediately before the last key addition in the encryption and comes immediately after the first key addition in the decryption. A cache latency 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 $d - 1$ 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. There are two different ways to compute key addition. The key can be added directly to the masked state and the key can also be masked for addition in the masked state.

A substantial amount of random bits is required to generate the masked blocks and to compute the masked `AND` described in Algorithm 1. We implemented random number generation through the standardized HASH_DRBG [Barker and Kelsey 2012] instantiated with the SHA-256 hash function. This choice proved to be faster than reading bytes from `/dev/urandom` by a 10-factor. Even if faster, generating random bits still impose a massive performance penalty and represents around 97% of the execution time in the masked implementation.

4. Experimental results

Our implementations were benchmarked in five different platforms:

- **Cortex-M3:** Arduino Due powered by an Atmel SAM3X8E ARM Cortex-M3 84MHz CPU. The compiler provided by the latest version of the Arduino Development Kit, GCC 4.8.4, was used with flags `-O3 -fno-schedule-insns -nostdlib -mcpu=cortex-m3 -mthumb`. 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 MK20DX256VLH7 Cortex-M4 72MHz processor. The same compiler used in the Cortex-M3 platform was used, but with flags `-O3 -fno-schedule-insns -nostdlib -mcpu=cortex-m4 -mthumb`. Execution time was measured through a native cycle counting register and some Assembly code.
- **Cortex-A15:** ODROID-XU4 board containing a Samsung Exynos5422 Cortex-A15 2Ghz and Cortex-A7 octa-core CPUs. We installed the official distribution of Arch Linux for the board, which comes equipped with GCC 6.1.1 for ARM, using the flags `-O3 -fno-schedule-insns -mcpu=cortex-a15 -mthumb -march=native`. 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 quad-core CPUs. We installed Arch Linux and employed GCC 6.1.1 cross-compiled for ARM with flags `-O3 -fno-schedule-insns -mcpu=cortex-a53 -mthumb -march=native`. 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 6.1.1 was again used with flags `-O3 -fno-schedule-insns -mssse3 -msse march=native`. The RDTSC register was used for cycle counting.

Table 1 presents 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 constant-time implementations in C receive the CT abbreviation. The vector implementations are intrinsically constant time by operating over registers only. Cycle counts were computed by encrypting *and* decrypting a 100 times the same message of length 1024 bytes and dividing the result by twice the number of bytes. The final result represents the average time to encrypt *or* decrypt a single byte using a specific implementation.

Table 1. Execution time and code size (ROM bytes) for Fantomas benchmarked in Cortex-M3/M4/A15/A53 ARM and Core i7 x86 Ivy Bridge. Figures present average cycles for encrypting or decrypting a single byte (CPB) in CBC/CTR mode, possibly using a constant-time implementation (CT) and vectorized implementation in NEON/SSE.

	Cortex-M3		Cortex-M4		Cortex-A15		Cortex-A53		I7 Ivy Bridge	
	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)
Fantomas 32 (CBC)	**177.44**	**3202**	143.62	3276	51.91	3900	87.65	3764	68.73	4040
Fantomas 32 (CTR)	**171.88**	**1916**	136.92	1934	50.71	2284	83.03	2364	67.06	2449
Fantomas 32 CT (CBC)	614.82	1858	489.13	1848	443.13	3084	232.73	3292	194.13	3680
Fantomas 32 CT (CTR)	629.59	1272	491.65	1262	443.44	1884	228.16	2148	194.13	2257
Fantomas 64 (CBC)	174.64	3198	142.88	3278	52.27	3628	80.87	3452	58.33	3736
Fantomas 64 (CTR)	168.81	1914	133.79	1934	50.66	2208	76.03	2200	56.03	2348
Fantomas 64 CT (CBC)	1920.06	4826	1559.42	4814	966.21	9372	226.54	2984	149.47	3395
Fantomas 64 CT (CTR)	1920.47	2738	1555.54	2734	961.77	5084	222.13	1984	127.95	2209
Fantomas NEON/SSE (CBC)	–	–	–	–	65.10	1844	62.04	1340	59.71	1490
Fantomas NEON/SSE (CTR)	–	–	–	–	63.81	1264	59.50	1160	46.49	1164
Fantomas16 NEON/SSE (CTR)	–	–	–	–	**16.07**	6846	17.93	3884	**5.95**	6139
Related work										
Fantomas 32 (CBC) Fast[1]	274.21	4620	–	–	–	–	–	–	–	–
Fantomas 32 (CTR) Fast[1]	220.13	2088	–	–	–	–	–	–		–
Fantomas 32 (CBC) Compact[1]	370.79	2916	–	–	–	–	–	–	–	–
Fantomas 32 (CTR) Compact[1]	520.94	1384	–	–	–	–	–	–	–	–
Fantomas16 NEON/SSE (CTR)[2]	–	–	–	–	26.60*	–	–	–	12.00*	–

[1] [Dinu et al. 2015] [2] [Grosso et al. 2015a] * adjusted timings

4.1. Discussion

The table contains plenty of interesting results to be discussed. 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, although more than 3 times slower than the unprotected version. If the main objective is to obtain a smaller code fingerprint and/or resistance against timing-based side-channel attacks, this implementation can still be a good choice. Observe that Cortex-A processors and even some Cortex-M4 microcontrollers have cache memory, so it is also important to measure the performance impact of protecting the implementations against cache-timing leakage. The 64-bit implementations of Fantomas showed little difference in comparison to the 32-bit code in all platforms, with the i7 Ivy Bridge processor as the only exception with a 22% speedup. The Cortex-A53 is a great platform for the 32-bit constant time implementation, producing an almost twice faster implementation than the Cortex-A15. On the other hand, 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. Our implementations were also tailored for ARM processors and enjoy the benefits of the second-operand barrel shifter as much as possible. Although not comparable, we note that cycle counts for 32-bit Fantomas were 25% lower in the Cortex-A15 than a Desktop machine equiped with the Ivy Bridge processor.

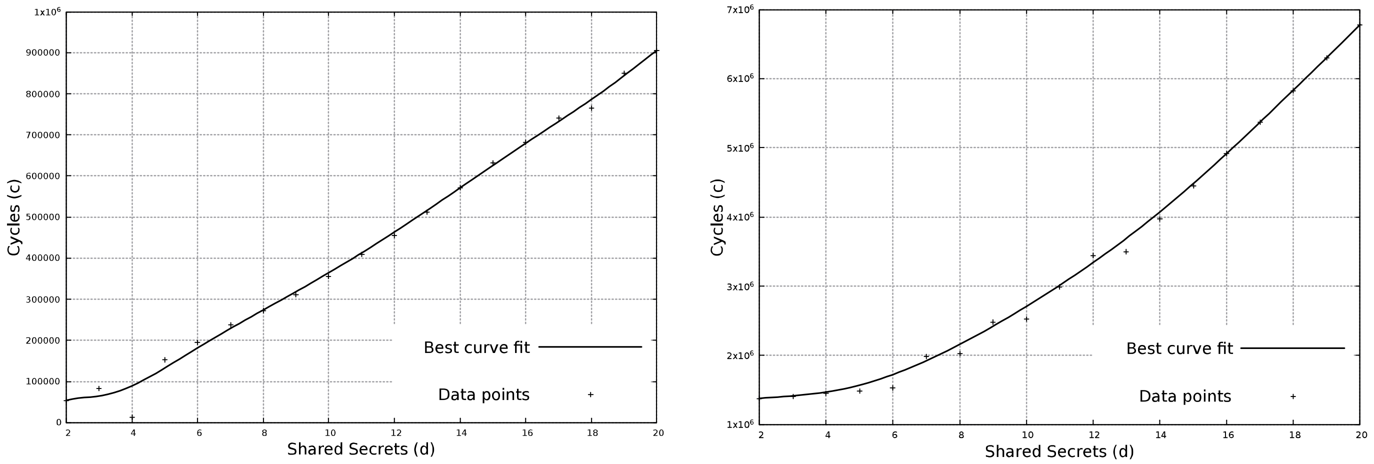
Code sizes generally grow from the Cortex-M3 to the Ivy Bridge. In the case of 64-bit code, there is a small code reduction starting in the Cortex-A53. This can be explained by the fact that the Cortex-M3/M4/A15 do not have 64-bit instructions, which means that 64-bit operations insert pairs of 32-bit instructions, resulting in a larger footprint. The code size for the Fantomas16 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 the similar SSE version in the Core i7. There is a clear space-time trade-off in this implementation. It is the largest implementation in terms of code size, but also the fastest among all platforms supporting vector instructions.

Cycle counts for the masked implementation of Fantomas are presented in Figure 7. A clear quadratic trend for the performance degradation can be observed in the graph, as expected. Two versions were implemented: masked key addition and conventional key addition. The two versions take such a close execution time that performance figures are essentially the same, so we opted to only present the masked key addition version together with the best curve fit. This happens because generating random bytes for the masked `AND` operations in the S-boxes consumes approximate 97% of the execution time. Additionally, the data point for a single shared secret is lower than the general trend. The reason for this is mostly the lower requirement of random bytes, although simple calls to the random number generation already cause a substantial performance penalty.

4.2. Comparison to related work

There are two main related works that established the previous state of the art in the context of this work. The most recent is the massive implementation effort from the FELICS framework [Dinu et al. 2015] to compare lightweight block ciphers performance-wise in representative platforms of 8, 16 and 32 bits. The project website [2] also contains

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



(a) Without random bit generator.

(b) With DRBG_HASH using SHA256.

Figure 7. Cycle counts for encrypting and decrypting with the masked implementation of Fantomas using masked key addition in the Cortex-A15 platform, as a function of the number d of shares.

results for some stream ciphers and block ciphers underlying MAC constructions. The target 32-bit 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. In the paper, the best implementation according to their Figure of Merit (FOM) takes 70,197 cycles using 4620 bytes of ROM (or 274.21 CPB in Table 1). The website has more recent numbers for an implementation capable of encrypting and decrypting in 94,921 cycles which consumes 2916 bytes of ROM (or 370.79 CPB in Table 1). Our implementation is 35.4% and 52.2% faster than their implementations, respectively, and competitive in terms of code size with the more compact implementation. 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 8335 cycles and 1384 bytes of ROM (520.94 CPB), the most efficient takes 3522 cycles and 2088 bytes of ROM (220.13 CPB) and a good trade-off is found at 4550 cycles and 2184 bytes of code size (284.38 CPB). After the proper conversions, our implementation improves these figures by 66.5%, 20.9% and 37.7%, respectively, by spending only 1916 bytes of ROM. As a reference point, FELICS reports much higher latencies for standardized block ciphers such as AES under different operating modes (73,868 cycles for encrypting and 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 [Grosso et al. 2015a]. 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 [Bernstein and Lange 2016]. By using the difference observed when benchmarking the SCREAMv3 reference code, we adjusted the timings in the CAESAR slides for architectures ARM Cortex-A15 by a factor of 1.5; and Intel Core i7 Ivy Bridge by a factor of 2, both observed in Table 1. After a simple comparison, we observed an approximate performance gain of 40% and 50% of our implementation when compared to the adjusted timings in the ARM and Intel platforms, respectively.

5. Conclusion

We presented several serial and vectorized software implementations of the Fantomas block cipher, producing more efficient and compact implementations in the ARM and x86 target platforms. Two approaches for side-channel resistance were implemented: constant time and masking. 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. The masked implementation 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 40 when compared to a constant time implementation. Highly-efficient random number generation is a paramount research target for enabling masked implementations in software to perform well in the target platforms.

Acknowledgements

The authors acknowledge support from LG Electronics Inc. during the development of this research, under the project "*Efficient and Secure Cryptography for IoT*".

References

Aciiçmez, O., Koç, c. K., and Seifert, J.-P. (2007). On the power of simple branch prediction analysis. In *ASIACCS*, pages 312–320. ACM.

Barker, E. and Kelsey, J. (2012). NIST SP 800-90A – Recommendation for Random Number Generation Using Deterministic Random Bit Generators.

Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., and Wingers, L. (2013). The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404. `http://eprint.iacr.org/2013/404`.

Bernstein, D. J. (2004). Cache-timing attacks on AES. URL: `http://cr.yp.to/papers.html#cachetiming`.

Bernstein, D. J. and Lange, T. (2016). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`.

Bonneau, J. and Mironov, I. (2006). *Cache-Collision Timing Attacks Against AES*, pages 201–215. Springer Berlin Heidelberg, Berlin, Heidelberg.

Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E. B., Knezevic, M., Knudsen, L. R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S. S., and Yalçın, T. (2012). *PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications*, pages 208–225. Springer Berlin Heidelberg, Berlin, Heidelberg.

Canteaut, A., Duval, S., and Leurent, G. (2016). *Construction of Lightweight S-Boxes Using Feistel and MISTY Structures*, pages 373–393. Springer International Publishing, Cham.

Daemen, J. and Rijmen, V. (2002). The - advanced encryption standard.

Dinu, D., Corre, Y. L., Khovratovich, D., Perrin, L., Großschädl, J., and Biryukov, A. (2015). Triathlon of lightweight block ciphers for the internet of things. Cryptology ePrint Archive, Report 2015/209. `http://eprint.iacr.org/`.

Grosso, V., Laurent, G., Standaert, F., Varici, K., Durvaux, F., Gaspar, L., and Kerckhof, S. (2015a). CAESAR candidate SCREAM Side-Channel Resistant Authenticated Encryption with Masking. `http://2014.diac.cr.yp.to/slides/leurent-scream.pdf`.

Grosso, V., Laurent, G., Standaert, F., Varici, K., Durvaux, F., Gaspar, L., and Kerckhof, S. (2015b). SCREAM Side-Channel Resistant Authenticated Encryption with Masking. `https://competitions.cr.yp.to/round2/screamv3.pdf`.

Grosso, V., Leurent, G., Standaert, F.-X., and Varıcı, K. (2015c). *LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations*, pages 18–37. Springer Berlin Heidelberg, Berlin, Heidelberg.

Hong, D., Lee, J.-K., Kim, D.-C., Kwon, D., Ryu, K. H., and Lee, D.-G. (2014). *LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors*, pages 3–27. Springer International Publishing, Cham.

Ishai, Y., Sahai, A., and Wagner, D. (2003). *Private Circuits: Securing Hardware against Probing Attacks*, pages 463–481. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer.

Kocher, P. C., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer.

Leander, G., Minaud, B., and Rønjom, S. (2015). A Generic Approach to Invariant Subspace Attacks: Cryptanalysis of Robin, iSCREAM and Zorro. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 254–283. Springer.

Piret, G., Roche, T., and Carlet, C. (2012). *PICARO – A Block Cipher Allowing Efficient Higher-Order Side-Channel Resistance*, pages 311–328. Springer Berlin Heidelberg, Berlin, Heidelberg.

Rivain, M. and Prouff, E. (2010). *Provably Secure Higher-Order Masking of AES*, pages 413–427. Springer Berlin Heidelberg, Berlin, Heidelberg.

Silva, B. R., Pereira, F. M. Q., and Aranha, D. F. (2016). Sparse representation of implicit flows with applications to side-channel detection. In *25th International Conference on Compiler Construction (CC)*, pages 110–120. ACM.