Hardware Implementation and Side-Channel Analysis of Lapin

Lubos Gaspar[1], Gaëtan Leurent[1,2], and François-Xavier Standaert[1]

[1] ICTEAM/ELEN/Crypto Group, Université catholique de Louvain, Belgium.
[2] Inria, EPI SECRET, Rocquencourt, France.
e-mails: {lubos.gaspar,fstandae}@uclouvain.be,gaetan.leurent@inria.fr

Abstract. Lapin is a new authentication protocol that has been designed for low-cost implementations. In a work from RFIDsec 2012, Berstein and Lange argued that at similar (mathematical) security levels, Lapin's performances are below the ones of block cipher based authentication. In this paper, we suggest that as soon as physical security (*e.g.* against side-channel attacks) is taken into account, this criticism can be mitigated. For this purpose, we start by investigating masked hardware implementations of Lapin, and discuss the gains obtained over software ones. Next, we observe that the structure of our implementations significantly differs from block cipher ones (for which most results in side-channel analysis apply), hence raising questions regarding how to evaluate physical security in this case. We then provide first results of side-channel analyzes against unprotected and masked Lapin. Despite interesting properties of the masked implementations, our conclusions are still contrasted because of the on-chip randomness requirements of Lapin protocol. These results give strong incentive to design similar but deterministic protocols, *e.g.* based on the recently introduced Learning With Rounding assumption.

Keywords: LPN, Ring-LPN, masking, side-channel analysis

1 Introduction

In [9], Heyse at al. proposed the Lapin authentication protocol based on the hardness of the Ring-LPN problem. Authors described two different Lapin variants based on a carefully chosen ring $\mathsf{R} = \mathbb{F}_2[X]/f(X)$. In the first variant, the ring is constructed with respect to an irreducible polynomial $f(X)$ in $\mathbb{F}_2$. This way the ring becomes a Galois field. In the second variant the polynomial $f(X)$ is reducible and it factors into distinct irreducible factors over $\mathbb{F}_2$, leading to improved performances (only this second variant will be considered next).

This work has been funded in part by the ERC project 280141 (acronym CRASH), by the European Commissions 7th framework program's project TAMPRES, and by the Belgian Cybercrime Center of Excellence for Training Research and Education (B-CCENTRE). F.-X. Standaert is an associate researcher of the Belgiam Fund for Scientific Research (FNRS-F.R.S).

The claim that such a protocol could provide better performances than standard solutions using block ciphers gave rise to some debate, as witnessed by the work of Bernstein and Lange [3]. In this paper, the authors strongly argued against Lapin, because of its unclear security level, and performances that are anyway below the ones of lightweight ciphers. In this paper, we aim to mitigate these criticisms in light of the interesting properties of Lapin regarding side-channel resistance. Namely, we would like to argue that[3] as the (physical) security level against side-channel attacks required by some application increases, Lapin gradually becomes an interesting alternative over the AES. The main reason of this interesting feature is the linearity found in its core operations.

For this purpose, we first propose a generic hardware architecture for Lapin, and detail the performance gains that can be obtained from its implementation in an FPGA (compared to previous software implementations of unprotected Lapin and masked AES). Next, we provide a preliminary evaluation of its side-channel properties. Interestingly, the situation of Lapin can be compared to recent investigations of randomness extractors against side-channel attacks [15]. Namely, they can both be masked quite efficiently, while raising questions regarding how to best exploit/evaluate side-channel leakage. As a first step in this direction, we suggest two ways to mount attacks against Lapin: one non-divide-and-conquer DPA-like attack, and one divide-and-conquer collision-like attack, exploiting the correlation between the leakage corresponding to multiple messages.

Overall, these results suggest that Lapin could be a promising candidate for (reasonably) lightweight and physically secure implementations. Yet, and admittedly, a significant drawback remains that it requires the generation of randomness on-chip, which may be an issue both from the performance and the physical security point of view. As the previous work in [9], we ignored this part of the problem so far, leading to two important questions for further research. First, how to generate this noise efficiently and in a leakage-resilient manner. Second, can we build an authentication protocol similar to Lapin, but deterministic, *e.g.* using the recently introduced Learning With Rounding assumption [1, 2].

2 Background

In this section we recall the Lapin authentication protocol and the masking countermeasure.

2.1 The Lapin protocol

Lapin is a two-round authentication protocol, illustrated in Figure 1. It is defined over the ring $\mathsf{R} = \mathbb{F}_2[X]/f(X)$, where f is a polynomial over $\mathbf{F}_2$ of degree n. The initial public parameters are: λ – security level parameter (in bits); π – mapping $\{0,1\}^\lambda \rightarrow \mathsf{R}$; $\tau \in (0, 1/2)$ – Bernoulli distribution parameter; $\tau' \in (\tau, 1/2)$ – reader acceptance threshold. Besides, the secret key of the tag and reader is defined as $K = (s, s')$, with $(s, s') \stackrel{\$}{\leftarrow} \mathsf{R}$. The protocol is executed as follows.

[3] Up to some limitations related to the randomness requirements of Lapin - see next.

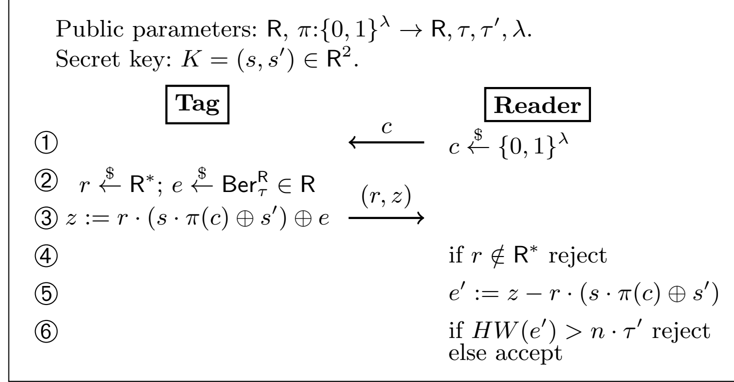

Fig. 1. Two-round Lapin authentication protocol.

After the tag is detected in the reader's vicinity, the reader randomly generates a challenge $c \in \{0,1\}^\lambda$ and sends it to the tag (step ① in Fig. 1). The λ parameter determines the security level of the protocol (*e.g.* $\lambda = 80$ bits). In the meantime, the tag generates parameters r and e (step ②). The parameter r is an uniformly chosen element of the ring R^* and e is a low-weight ring element chosen with Bernoulli distribution over $\mathbb{F}_2$ (Ber_τ) with parameter (bias) $\tau \in]0, 1/2[$ (*i.e.*, $\Pr[X = 1] = \tau$ if $X \leftarrow \mathsf{Ber}_\tau$). After receiving the challenge c, the tag maps the challenge to the ring through π, where π satisfies $\pi(c) \oplus \pi(c') \in \mathsf{R} \setminus \mathsf{R}^* \Leftrightarrow c = c'$. We denote R^* the set of elements in R that have a multiplicative inverse. Subsequently, the tag responds with $(r, z = r \cdot K(c) \oplus e) \in \mathsf{R} \times \mathsf{R}$ (step ③), where $K(c) = s \cdot c \oplus s'$ is the session key that depends on the shared secret key $K = (s, s') \in \mathsf{R}^2$ and the challenge c. The reader accepts if $e' = z \oplus r \cdot K(c)$ (computed in the step ⑤) is a polynomial of low weight (step ⑥). More details on the Lapin protocol and all necessary security proofs can be find in [9].

Chinese Reminder Theorem representation (CRT): In this work, we focus on versions of Lapin over a ring $\mathsf{R} = \mathbb{F}_2[X]/f(X)$ where $f(X)$ factors into *distinct* irreducible factors over $\mathbb{F}_2$. For an element h in the ring $\mathbb{F}_2[X]/f(X)$, we will denote $\widehat{h}$ its CRT representation with respect to the factors of $f(X)$ (for simplicity $f(X)$ will be further denoted only as f). In other words, if $f = f_1 \cdot f_2 \cdots f_m$ where all f_j are irreducible, then:

$$\widehat{h} \doteq (h \bmod f_1, \ldots, h \bmod f_m), \qquad \widehat{h_i} \doteq h \bmod f_i.$$

For the protocol to be implemented efficiently, all public and private values must be transformed to the CRT domain. However, in order to obtain the resulting tag response (r, z), it must be reconstructed from the response $(\widehat{r}, \widehat{z})$ as follows:

$$(r, z) = \left(\bigoplus_{i=1}^{m} \widehat{r}_i \cdot \overbrace{\frac{f}{f_i} \cdot \left[\left(\frac{f}{f_i} \right)^{-1} \right]_{f_i}}^{\text{constant}}, \bigoplus_{i=1}^{m} \widehat{z}_i \cdot \overbrace{\frac{f}{f_i} \cdot \left[\left(\frac{f}{f_i} \right)^{-1} \right]_{f_i}}^{\text{constant}} \right). \qquad (1)$$

Although constants in the equation can be precomputed, this transformation still involves m multiplications and additions of size n. Since the transformation from and to the CRT representation only uses public values, the tag response (r, z) can be sent to the reader in its CRT representation $(\widehat{r}, \widehat{z})$ without decreasing Lapin's security. This way, the computationally extensive transformation can be performed at the reader side.

The challenge mapping π is defined as $\widehat{\pi(c)} = (c, c, c, c, c)$, *i.e.* each CRT component is just the challenge padded with zeroes.

2.2 The masking countermeasure

Masking is a countermeasure against power analysis attacks based on secret sharing, first proposed by Chari et al. [5] and Goubin et al. [7]. Its main objective is to decrease the correlation between the power consumed by a device and the data being processed, by applying one (or several) random mask(s) to intermediate values. More formally, prior to the execution of the algorithm, all sensitive values (*i.e.* all key-dependent intermediate results used during the cryptographic computations) must be split into shares. Next, the algorithm is implemented in such a way that the processing is only performed on these shares, which are recombined at the end of the computation to produce the correct output result. Given that the shares are refreshed for each new authentication[4], masking provides an increase of the side-channel attacks data complexity that is exponential in their number, under the assumption that the leakage of each share is independent of the others. However, for this exponential security increase to materialize into strong concrete security, it is required that sufficient noise is present in the leakage measurements [18].

Different types of masking schemes have been proposed in the literature. Boolean masking (where the sharing is performed using a bitwise XOR operation) appears as the most natural candidate in our context, since it can take advantage of the linearity of the computations in Lapin. In this context, the split of a sensitive value h into d shares requires the generation of $d-1$ random mask values q_i, and is defined as follows:

$$h_1 = q_1, \qquad \ldots, \qquad h_{d-1} = q_{d-1}, \qquad h_d = h \oplus \bigoplus_{i=1}^{d-1} q_i.$$

Based on this sharing, a masked version of Lapin becomes straightforward to implement. The secret keys s, s' and low-weight element e are first divided to shares $s_1, s_2, \ldots, s_d$; $s'_1, s'_2, \ldots, s'_d$ and $e_1, e_2, \ldots, e_d$, respectively. The final result z is then obtained by recombining shares $z_1, z_2, \ldots, z_d$ that are computed as

[4] This implies storing all the shares of the secret key, for which the initial split is assumed to be performed once without leakage (otherwise this initialization can always be the target of simple attacks).

follows:

$$\begin{aligned} z &= (\pi(c) \cdot s \oplus s') \cdot r \oplus e, \\ &= [\pi(c) \cdot (s_1 \oplus \cdots \oplus s_d) \oplus (s'_1 \oplus \cdots \oplus s'_d)] \cdot r \oplus (e_1 \oplus \cdots \oplus e_d), \\ &= [(\pi(c) \cdot s_1 \oplus s'_1) \cdot r \oplus e_1] \oplus \cdots \oplus [(\pi(c) \cdot s_d \oplus s'_d) \cdot r \oplus e_d], \\ &= z_1 \oplus \cdots \oplus z_d, \end{aligned}$$

with

$$z_i \doteq (\pi(c) \cdot s_i \oplus s'_i) \cdot r \oplus e_i.$$

Since Lapin is linear, we can compute the shares z_i independently, and there is no need of interaction between them nor refreshing during the computations, as opposed to masking non-linear gates within the AES [16], for instance. This leads to very efficient implementations.

On the independent leakage assumption: In addition, the linearity of Lapin also allows to compute the shares sequentially. This time separation typically reduces the risk of glitches and other hardware effects that are well known to contradict the independence assumption [13]. This is especially interesting in the context of hardware implementations as considered in the next section; this is the typical context in which glitches can appear [14].

3 Hardware implementation

In this section we discuss our design choices for (unprotected and masked) Lapin. We present several implementations of a Lapin co-processor and report their area and timing performance. A hardware implementation takes advantage of parallel computing in order to generate sufficient algorithmic noise, which allows significant security gains in practice (and improved performances).

3.1 Generic architecture

In order to implement the Lapin protocol, we use the same parameter values as defined in Heyse at al. [9]. The degree of the polynomial f is chosen as $n = 621$, the security level parameter $\lambda = 80$ bits, Bernoulli distribution bias parameters $\tau = 1/6$, $\tau' = 0.29$ and the number of factors of f as $m = 5$. The five f_j polynomials are defined as follows:

$$\begin{aligned} f_1(X) &= X^{127} \oplus X^8 \oplus X^7 \oplus X^3 \oplus 1, \\ f_2(X) &= X^{126} \oplus X^9 \oplus X^6 \oplus X^5 \oplus 1, \\ f_3(X) &= X^{125} \oplus X^9 \oplus X^7 \oplus X^4 \oplus 1, \\ f_4(X) &= X^{122} \oplus X^7 \oplus X^4 \oplus X^3 \oplus 1, \\ f_5(X) &= X^{121} \oplus X^8 \oplus X^5 \oplus X \oplus 1. \end{aligned}$$

Assuming that the Lapin protocol is performed on d shares (each of them computed for all m CRT parts), we will denote the $i-th$ share of the $j-th$ CRT part

for a sensitive variable h as $\widehat{h_{i,j}}$. Next, we propose a flexible architecture that allows splitting the sensitive variables into arbitrary number of shares. Taking advantage of generic VHDL coding enables the generation of such implementations by re-synthetizing the same code with different parameters. For this purpose, we present the masked Lapin algorithm, the combined polynomial multiplication-reduction algorithm, and the hardware implementation of the complete Lapin core.

Masked Lapin: The generic masked Lapin algorithm is illustrated in Algorithm 1. First, a padded public challenge $\pi(c)$ is multiplied by a secret key s divided into shares $\widehat{s_{i,j}}$ for $1 \leq i \leq d$, $1 \leq j \leq m$. Following, the result is added to the secret key s' divided into shares $\widehat{s'_{i,j}}$. The sum is then multiplied by a public random tag response $\widehat{r_j}$. Subsequently, the product is added to a low-weight element $\widehat{e_{i,j}}$. The last step is to sum all resulting shares to form an unmasked tag response $\widehat{z_j}$. Finally, $\widehat{r_j}$ and $\widehat{z_j}$ are sent back to the reader to finish the authentication process. Note that all computations are performed on all m CRT parts.

Algorithm 1 Masked Lapin algorithm

Input:
1: Padded public challenge $\pi(c)$
2: Public random element $\widehat{r}$
3: Secret keys s and s' divided to shares $\widehat{s}$ and $\widehat{s'}$ respectively
4: Secret low-weigh error element e divided into shares $\widehat{e}$

Output: Response $(\widehat{z}, \widehat{r})$
5: **for** j from 1 to m **do**
6: $\quad \widehat{z_j} \leftarrow 0$
7: $\quad$ **for** i from 1 to d **do**
8: $\quad\quad \widehat{t_{i,j}} \leftarrow (\pi(c) \cdot \widehat{s_{i,j}} \oplus \widehat{s'_{i,j}}) \cdot \widehat{r_j} \oplus \widehat{e_{i,j}}$
9: $\quad$ **end for**
10: $\quad$ **for** i from 1 to d **do**
11: $\quad\quad \widehat{z_j} \leftarrow \widehat{z_j} \oplus \widehat{t_{i,j}}$
12: $\quad$ **end for**
13: **end for**
14: Return $\widehat{z}$

Reduction of a low-weight error element e: Unlike the other parameters in Lapin, the low-weight error element e cannot be generated or pre-stored in CRT representation directly: its m CRT parts must be calculated prior to other computations. The reduction of such a large element is not straightforward and requires additional hardware resources. In order to simplify this problem, we first write each share of e (a polynomial of degree 621) in Horner form using five polynomials $[e^{(4)}, e^{(3)}, e^{(2)}, e^{(1)}, e^{(0)}]$ of degree 127 (except for $e^{(4)}$ of degree

109):

$$\widehat{e_{i,j}} = \left(\left(\left(\left(e_i^{(4)} \cdot X^{128} \oplus e_i^{(3)}\right) \cdot X^{128} \oplus e_i^{(2)}\right) \cdot X^{128} \oplus e_i^{(1)}\right) \cdot X^{128} \oplus e_i^{(0)}\right) \bmod f_j. \quad (2)$$

Next, the polynomial X^{128} can be reduced by each characteristic polynomial f_j resulting in constant polynomials g_j of degree less than $\deg(f_j)$. After substitution Equation 2 becomes:

$$\widehat{e_{i,j}} = \left(\left(\left(e_i^{(4)} g_j \bmod f_j \oplus e_i^{(3)}\right) \cdot g_j \bmod f_j \oplus e_i^{(2)}\right) \cdot g_j \bmod f_j \oplus e_i^{(1)}\right) \cdot g_j \bmod f_j \oplus e_i^{(0)}. \quad (3)$$

This way, only four multiplications, four reductions and four additions have to be computed to obtain each $\widehat{e_{i,j}}$. Moreover, the same hardware as used for performing the computations in Algorithm 1 can be re-used to calculate the $\widehat{e_{i,j}}$'s. Note that since $e^{(3)}$, $e^{(2)}$, $e^{(1)}$ and $e^{(0)}$ are of degree 127, some extra hardware is still necessary for their reduction.

Polynomial multiplication and reduction: Examining the previous algorithms reveals that $6 \times d \times m$ polynomial multiplications, reductions and polynomial additions have to be performed to generate a response $\widehat{z}$. Among those, the most time-consuming operations are the polynomial multiplications and subsequent reductions of the products. Although the performances of the "schoolbook" multiplication algorithm is theoretically lower ($\mathcal{O}(2^n)$) than the Karatsuba algorithm ($\mathcal{O}(n^{log_2 3})$), it has a very simple structure, and so the resulting implementation is area-efficient and can operate at high clock frequencies. Moreover, polynomial reduction and multiplication operations can be executed simultaneously in this case, so that no computational time is lost for the reduction step. For this reason, we have implemented a combined polynomial multiplication-reduction based on the schoolbook multiplication, as explained in Algorithm 2. Two input polynomials $\widehat{a_{i,j}}$ and $\widehat{b_{i,j}}$ of degree at most $n_j - 1$ (represented with bit arrays $A[n_j - 1 : 0]$ and $B[n_j - 1 : 0]$) are multiplied together while partial products are reduced by the characteristic polynomial f_j of degree n_j (represented with the bit array $F[n_j : 0]$) simultaneously. A closer examination of the algorithm shows that B is multiplied by one bit of A at a time. Therefore, if A contains a secret value, Lapin will be vulnerable to a Simple Power Analysis (SPA) attack, where each partial multiplication leaks one bit of a key. For this reason, A must contain only public data. On the contrary, B is processed in larger blocks (according to the datapath size), so we used it to manipulate sensitive data (that will additionally be protected against DPA thanks to masking).

Lapin architecture: We implemented Lapin as a hardware co-processor core, synthesized using Xilinx ISE 12.4 for Xilinx Virtex-5 XC5VLX50T FPGAs. Our implementation is illustrated in Figure 2. All variables are stored in the data register that is implemented in a dual-port embedded RAM. Random ring elements $\widehat{r}$ and low-weight error elements e have to be generated by a TRNG. Three

Algorithm 2 Combined polynomial multiplication-reduction

Input:
1: polynomial $\widehat{a_{i,j}}$, deg $(\widehat{a_{i,j}}) \leq n_j - 1$ represented as bit array $A[n_j - 1 : 0]$
2: polynomial $\widehat{b_{i,j}}$, deg $\left(\widehat{b_{i,j}}\right) \leq n_j - 1$ represented as bit array $B[n_j - 1 : 0]$
3: characteristic polynomial $f_j(X)$ of degree n_j represented as bit array $F_j[n_j : 0]$
Output: $c(X) = \left(\widehat{a_{i,j}}.\widehat{b_{i,j}}\right) \bmod f_j)$
4: $C \leftarrow 0$
5: **for** i from 1 to n_j **do**
6:   **if** $A[n_j - i] = 1$ **then**
7:     $C \leftarrow C \oplus B$
8:   **end if**
9:   **if** $C[n_j - 1] = 1$ **then**
10:     $C \leftarrow (C \ll 1) \oplus F_j[n_j - 1 : 0]$
11:   **else**
12:     $C \leftarrow (C \ll 1)$
13:   **end if**
14: **end for**
15: Return $C[n_j - 1 : 0]$

distinctive parts can be identified in the datapath of this Lapin core: the polynomial multiplication logic (shown in blue in Figure 2), the reduction logic (in red) and the addition logic (in green). During multiplication, a public parameter is stored in the shift register (implemented in logic). By shifting this register, one bit is selected at a time and multiplied with the secret parameter. The resulting partial product is stored in the accumulator (also implemented in a dual-port RAM). Subsequently, this partial product is shifted and added to the next partial product. Whenever the size of this sum exceeds n_j bits ($n_j = \deg(f_j)$), reduction circuitry is activated in the next clock cycle to reduce the exceeding bit. Once the multiplication is finished, the result can be summed with a next secret parameter stored in the data register. Prior to this addition, all exceeding bits of this parameter are reduced by an auxiliary reduction circuitry. As can be observed from the figure, multiplication involves shifting of partial products stored in the accumulator. However, if a partial product of size n_j is shifted in more than one clock cycle (which is the case when $k < 128$), the most significant bit of each shifted word must be stored in a carry bit register Cr. This way a stored carry bit becomes a least significant bit of the next word in the next clock cycle.

3.2 Performance evaluation

Implementation results: In order to investigate the performance trends resulting from different datapath sizes, we implemented our design for different values of the k parameter in Figure 2, namely we considered k =8-, 16-, 32-, 64- or 128-bit wide architectures, as summarized in Table 1. These results correspond to an unprotected implementation (*i.e.* $d = 1$) – but thanks to the

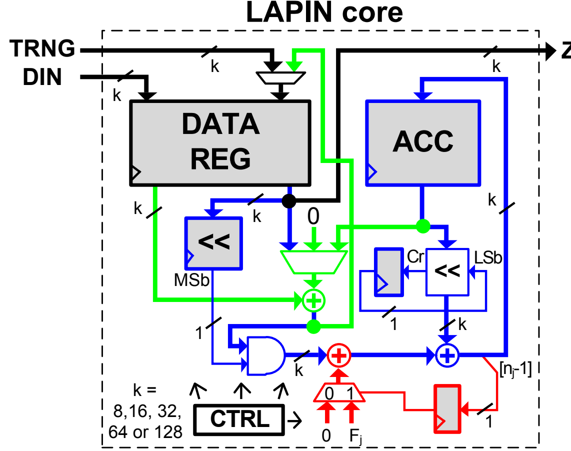

Fig. 2. Datapath with multiplication (blue), reduction (red) and addition (green) circuitry.

linear structure of Lapin, the masked versions have essentially the same cost: only the memory requirements will increase proportionally to the number of shares, in order to store intermediate results. We observe that if the datapath size is decreased by half, the number of allocated fine-grained FPGA resources does not always decrease accordingly. This can be explained by the fact that narrower datapaths usually require more multiplexers and more complex control logic. Moreover, this extra logic increases the overall datapath delays, resulting in lower maximal clock frequency (see the fifth column in Table 1).

Table 1. Implementation results: resource usage and timing information.

Datapath (k)	Slices	BRAM 18kb	BRAM 36kb	$f_{\max}$ (MHz)	Clock cycles $d=1$	Clock cycles $d=2$	Clock cycles $d=3$
8	213	2	0	125.3	20,977	41,969	62,961
16	232	2	0	127.5	10,489	20,985	31,481
32	311	1	1	127.2	5,245	10,493	15,741
64	330	0	3	130.2	2,623	5,247	7,871
128	451	0	6	140.3	1,332	2,664	3,996

Timing results: The detailed timing characteristics of our implementations are given in Table 2 (see Appendix B), in which each line corresponds to the number of clock cycles required for the computations in one characteristic polynomial domain (f_j), and the last line represents the total number of clock cycles for the full Lapin execution, *i.e.* one tag response for one authentication request.

The left part of the table shows results if no masking is used (secret variables are not divided into shares, *i.e.* $d = 1$); its right part summarizes results for three-share computations (*i.e.* $d = 3$). For completeness, timing characteristics are again provided for all the aforementioned datapath sizes. The datapath and the control logic were designed in order to eliminate cycles with no activity (*i.e.* pipeline bubbles). As a result, decreasing the datapath size by half results in doubling the number of clock cycles in most cases. The only exception is the 128-bit datapath where some extra dummy cycles were necessary to avoid RAM read/write collisions.

Comparison: The timing comparison of software AES [16], software Lapin [9] and our hardware Lapin are given in Figure 3. In the case of software Lapin, the cycle counts for the masked versions are extrapolated from the unprotected implementation. These results lead to two main observations.

First, we see that masked Lapin implementations indeed become interesting alternatives over AES ones, as the number of shares increases. This is caused by the fact the the implementation cost of non-linear operations (which become dominant in masked AES implementations) increases quadratically with d, while this increase is only linear in the case of Lapin. Interestingly, the number of shares for which this gain concretely appears is reasonably small, hence close to practical interest. The software figures we have for protected implementations of AES and Lapin on ATMega suggest that Lapin could become more efficient than AES even with $d = 2$, but the crossing point can move significantly depending on the masking scheme used, and the optimization level of the implementation.

Second, we see that (as usual) specialized hardware implementations allow a significant optimization of the performances of Lapin. Gains already appear in the comparison of 8-bit architectures: computing a tag response in software requires 112,500 clock cycles (that can be decreased to 30,000 clock cycles if precomputation is allowed); our 1-share 8-bit hardware implementation requires only 20,977 clock cycles in this case (without precomputations). These advantages naturally amplify as we consider larger datapath sizes.

# of shares d	AES *softw.* [16, 8]	Lapin *softw.* [9]	Lapin *8b hardw.*
1	5100	112500	20977
2	286844	225016	41969
3	572069	337532	62961
4	1003154	450048	83953
5	1489539	562564	104945
6	2095756	675080	125937
7	2779561	787596	146929

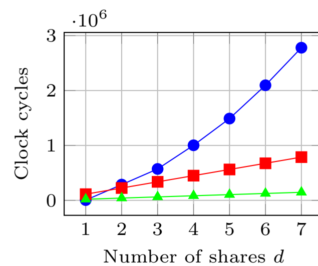

Fig. 3. Number of clock cycles vs. number of shares (d) for software AES [16, 8], software Lapin [9] and hardware Lapin. With increase of used shares, the computation time increases quadratically for the AES and only linearly for both Lapin implementations.

4 Side-channel analysis of Lapin

The previous section suggests that Lapin is an interesting candidate for masking. First, its linearity allows increasing the number of shares for only a linear implementation cost penalty. Second, it also allows manipulating the shares independently, which implies a better chance to fulfill the independent leakage requirements that is crucial for masking to provide its expected security improvements. Third, it is efficiently implemented in hardware with large datapaths, providing algorithmic noise that is needed for the exponential data complexity increase of masking to materialize into strong security levels. On the other hand, a limitation of this analysis remains that it "only" considers the security orders of the masking schemes (*i.e.* the minimum number of shares of which the leakage must be exploited to recover key-dependent information). While this is a traditional approach in side-channel analysis, it remains to understand how these security orders translate into actual attack complexities. In particular, since Lapin has a significantly different structure than block ciphers (to which most published higher-order side-channel attacks apply), it is interesting to study attacks that exploit the design of its multiplier. In this section, we consequently suggest several scenarios to analyze/evaluate a Lapin implementation using side-channel information, and we study the efficiency of those attacks depending on the masking order of this implementation. As will be seen, these attacks differ from classical DPA in some interesting respects.

We first point out that we can attack the CRT components independently: each component is computed separately, and we can test a key candidate $\widehat{s_i}$ from the an authentication transcript without knowing the other CRT components. In the following we describe attacks on a single CRT component.

As a starting point, we consider a non-protected implementation of Lapin, and we study how to apply a standard DPA attack. In order to evaluate power analysis on our implementation of Lapin, we first have to study what parts of the computations are key-dependent, and how it might affect the power consumption. In our analysis we target the first multiplication $s \cdot \pi(c)$ in the Lapin protocol, where s is a secret value, and c is a challenge that can be set by the attacker. Our architecture for Lapin includes a large accumulator that is updated at each clock cycle, as shown in Figure 2, and we assume that this accumulator will induce a significant leakage dependent on its value a. In order to simplify our analysis, we use a Hamming weight model, *i.e.* we assume that the power consumption is correlated with $\mathrm{HW}(a)$, and we run simulations where the samples are computed as $\mathrm{HW}(a) + N$, with N a Gaussian-distributed random noise. Our attacks will typically exploit the fact that, when computing $a \cdot c$, the multiplication algorithm updates the accumulator a as (we consider the optimized multiplication with an 80-bit c):

$$a_0 = 0 \qquad\qquad a_{i+1} \leftarrow \begin{cases} 2 \cdot a_i \oplus s & \text{if } c[80 - i] = 1 \\ 2 \cdot a_i & \text{otherwise.} \end{cases}$$

Hence, the value of the a after a few cycles of computation is a small multiple of the secret:

$$a_{80} = s \cdot c \qquad\qquad a_i = s \cdot \sum_{j=1}^{i} c[80-j]X^{i-j}.$$

Cautionary note: Assuming Hamming weight leakages is admittedly a simplification of the real measurements used in side-channel attacks. However, it is a reasonable abstraction for preliminary analyses, that has been used in numerous contexts [12]. While the actual complexities provided by these simulated attacks are only meaningful up to the extent that true leakages behave similarly, they are usually informative to confirm whether some attack techniques can be successful. This is typically what the following results aim to exhibit, *i.e.* how side-channel attacks against Lapin differ from standard DPA against block ciphers.

4.1 A first DPA-like attack against unprotected Lapin.

In an unprotected design, the leakage reveals the Hamming weight of multiples of the secret, with a chosen multiplier $m_i(c) = \sum_{j=1}^{i} c[80-j]X^{i-j}$, depending on the challenge c and the cycle i we target. If we exploit several different cycles in a given trace, we can get information about $\mathrm{HW}(a_i) = \mathrm{HW}(s \cdot m_i(c))$ for the same c and different values of i. However, the same information can also be obtained by targeting a fixed cycle ι of the computation if we capture several traces and send the appropriate challenges c_j so that $m_\iota(c_j) = m_j(c)$.

In a DPA attack, we guess a small part of the key, then predict the value of the leakage for a key guess according to a model, and compare the prediction to the actual measurements in order to rank the key candidates. For a block cipher, the key is usually divided according to the structure of the cipher; for instance, an attack on the AES will target the key bytes independently because each SBox in the first round depends on a single key byte. In the case of Lapin there is no such natural division of the key, but we can study the key bits required to compute some bits of the accumulator.

Recovering a few key bits. For a given t, if $m_i(c)$ is of degree at most t (*e.g.* if $i \leq t$), we can compute the p *least* significant bits of $s \cdot m_i(c)$ from the p *least* significant and the $t-1$ *most* significant bits of s. This allows to build a simple DPA attack: after guessing the key bits, we compute the least significant bits of a_i and we consider the remaining bits as algorithmic noise. We can then compute the correlation between the leaked weight of a and the weight of the predicted bits, and use it to rank the key candidates.

Note that if there is no measurement noise, we can only use 2^t different measures in this attack, because there are only 2^t different polynomials of degree t or lower. Extra measures are only useful to reduce the measurement noise.

We implemented this attack using Pearson's correlation coefficient as a comparison tool [4], and we show an example of results in Figure 8 in Appendix A.

We can see from this example that the information from the measures is not sufficient to recover exactly the secret key bits; there is a cluster of key candidates with the same correlation coefficient. This is due to the algebraic structure of the multiplier; for instance, if we consider two key candidate s and $s' = s \cdot X$, our prediction for the least significant bit of the accumulator using the key s will match the second-least significant bit of the accumulator for candidate s'. Therefore, both candidates will be in the same cluster.

Recovering the full key. As opposed to a typical side channel attack on a block cipher, we don't have independent parts of the key affecting different parts of the computation. Therefore, we don't attack key parts independently using a divide-and-conquer approach, but we recover key information gradually: if we have a good candidate for n bits of the key, we generate key candidates for $n+1$ bits by considering both values for the next key bit. This defines a tree of key candidates, and we explore the tree following the best candidates.

More precisely, we compute a score for the candidates as the ratio of Pearson's correlation coefficient over the expected correlation coefficient for the right key (the square root of the number of predicted bits divided by the total number of bits on the bus, over which the Hamming weight is computed). This allows to compare the quality of key guesses of different lengths: if more key bits are guessed, we can predict more bits, and we expect a better correlation coefficient. The score of a node is computed when its parent is explored, and we select the node with the higher score among all nodes whose score has been computed. In practice, we use a priority queue to store the nodes and to extract the best one efficiently.

If we have 2^t traces, we begin by guessing the $t-1$ most significant bits, and one least significant bit of the key; this allows to predict one bit of the accumulator a_t after t cycles. Next, we guess the second-least significant bit of the key, so that we can predict two bits of a_t.

Figure 5 in Appendix A shows the success rate of this DPA-like attack, with various parameters. The attack becomes less efficient with a large datapath, because the Hamming weight over a larger bus size introduces more algorithmic noise to the predicted value.

Those experiments clearly show the effect of the algorithmic noise from the unknown bits in the Hamming weight (with variance $k/4$ for a k-bit datapath), and of the physical noise (with variance σ^2). When the physical noise is dominant, *i.e.* $\sigma^2 > k/4$, we see the data complexity increasing linearly with the variance of the noise, as expected [11]. For instance, our experiment show a data complexity of about $2^7.\sigma^2$ to reach a high success rate with $k = 8$. When k increases, this increases the algorithmic noise, and we have a similar increase of the data complexity if the algorithmic noise is dominant, *i.e.* when $\sigma^2 < k/4$. We note that this increase is somewhat faster than $k/4$ because there are more nodes to explore to locate the correct key when k is larger, but we stop after a fixed number of nodes are explored.

4.2 Collision-like attack.

We now describe an attack based on the structure of the operations in Lapin. The main advantage of this attack is that we can eliminate the algorithmic noise due to the Hamming weight with a large datapath by comparing the leakage with two different inputs. This is similar to side-channel collision attacks [17] where two traces are compared to detect specific events.

More precisely, we use the fact that the operation $\alpha \mapsto \alpha \cdot X$ has a predictable effect on the Hamming weight of α. We have:

$$\alpha \cdot X \bmod f = \begin{cases} (\alpha \ll 1) & \text{if } \mathrm{MSB}(\alpha) = 0 \\ (\alpha \ll 1) \oplus f & \text{if } \mathrm{MSB}(\alpha) = 1, \end{cases}$$

where $\ll$ to denotes a left shift. Alternatively, we can write it using a rotation $\lll$ over $\deg(f)$ bits:

$$\alpha \cdot X \bmod f = \begin{cases} (\alpha \lll 1) & \text{if } \mathrm{MSB}(\alpha) = 0 \\ (\alpha \lll 1) \oplus \bar{f} & \text{if } \mathrm{MSB}(\alpha) = 1, \end{cases}$$

where $\bar{f} = f \oplus X^{\deg(f)} \oplus 1$ is f without the highest and lowest coefficients. Since the polynomials f used in Lapin are pentanomials, we have $\mathrm{HW}(\bar{f}) = 3$, and we can relate the Hamming weight of α and the Hamming weight of $\alpha \cdot X \bmod f$:

$$\mathrm{HW}(\alpha \cdot X \bmod f) = \begin{cases} \mathrm{HW}(\alpha) & \text{if } \mathrm{MSB}(\alpha) = 0 \\ \mathrm{HW}(\alpha) + 3 & \text{if } \mathrm{MSB}(\alpha) = 1 \text{ and } \mathrm{HW}(\alpha \lll 1 \wedge \bar{f}) = 0 \\ \mathrm{HW}(\alpha) + 1 & \text{if } \mathrm{MSB}(\alpha) = 1 \text{ and } \mathrm{HW}(\alpha \lll 1 \wedge \bar{f}) = 1 \\ \mathrm{HW}(\alpha) - 1 & \text{if } \mathrm{MSB}(\alpha) = 1 \text{ and } \mathrm{HW}(\alpha \lll 1 \wedge \bar{f}) = 2 \\ \mathrm{HW}(\alpha) - 3 & \text{if } \mathrm{MSB}(\alpha) = 1 \text{ and } \mathrm{HW}(\alpha \lll 1 \wedge \bar{f}) = 3. \end{cases}$$

Therefore, the distribution of $\mathrm{HW}(\alpha \cdot X) - \mathrm{HW}(\alpha)$ for a random α is the following:

if $\mathbf{MSB}(\boldsymbol{\alpha}) = \mathbf{0}$: $\mathrm{HW}(\alpha \cdot X) - \mathrm{HW}(\alpha) = 0$,

if $\mathbf{MSB}(\boldsymbol{\alpha}) = \mathbf{1}$: $\mathrm{HW}(\alpha \cdot X) - \mathrm{HW}(\alpha) = \begin{cases} +3 & \text{with probability } 1/8 \\ +1 & \text{with probability } 3/8 \\ -1 & \text{with probability } 3/8 \\ -3 & \text{with probability } 1/8. \end{cases}$

To exploit this property, we will use two measures such that $m_i(c) = m$ and $m_{i'}(c') = m \cdot X$. Then, we can recover the value $\mathrm{MSB}(m \cdot s)$ (*i.e.* a linear equation in s) by comparing $\mathrm{HW}(m \cdot s)$ and $\mathrm{HW}(m \cdot X \cdot s)$ (we use the analysis above with $\alpha = m \cdot s$). If there is no noise, we will recover a key bit with only two measures, with probability one.

As opposed to the attack of Section 4.1, this analysis uses the full state of the multiplier, and avoids algorithmic noise due to the Hamming weight. This makes the attack quite efficient. However, there is also an important limitation:

because the challenge used in Lapin is only 80-bit long, the multiplication $m \cdot c$ only takes 80 cycles, and we can only recover 80 bits from each CRT component of the key with this technique.

If there is some measurement noise, we can remove it either by repeating the measures of $\mathrm{HW}(m \cdot s)$ and $\mathrm{HW}(m \cdot X \cdot s)$ and averaging them, or by using all the measures in a template attack [6]. We performed simulations with various levels of noise using a template attack, and the rank estimation code from [19] to compute the rank of the full key from the estimated key bits probability. We report our results in Appendix A, Figure 6. Those experiments show that with a 128-bit datapath we can recover 80 key bits with very few candidates using only 2^6 traces with a noise variance of 1. Again, the data complexity grows linearly with the noise variance σ^2, and we need about $2^6 \cdot \sigma^2$ traces to reduce the key space to a few candidates.

If the datawidth k is smaller than 128, we have to combine $128/k$ measures to build the full Hamming weight $\mathrm{HW}(a)$ in order to perform the attack. If the noise variance is σ^2 this becomes equivalent to a noise variance of $128/k \cdot \sigma^2$ for an attack with a 128-bit datapath, and the number of traces required is about $2^6 \cdot \sigma^2 \cdot 128/k$. As expected, this behavior is opposite to what happens in the attack of Section 4.1 where a larger datapath implies a higher attack complexity because of the extra algorithmic noise.

For the acquisition of the data, one can either extract the two leakages from a single trace at two different points of interest, or use two traces with chosen challenges and extract the leakage from each trace at the same point in time. In order to minimize the number of traces required for the attack, we use all the clock cycles of the multiplier. More precisely, we send the challenge $c = 2^{79}$, so that $m_i(c) = X^{i-1}$ and $m_{i+1}(c) = X^i = m_i(c) \cdot X$.

Order of the attack. This attack exploits information from two different measures, and combines them using the difference operation. Since we use a single pair of challenges for each key bit, we can average the measures and the information can be extracted from the average leakage values by testing whether it is zero or in $\{-3, -1, +1, +3\}$. Therefore, this attack can be seen as a first-order bivariate attack.

4.3 Attack on masked Lapin

We now study how this attack can be applied against a masked implementation of Lapin such as the implementation described in Algorithm 1. In a masked implementation the multiplication $\pi(c) \cdot s$ is split in d computations $\pi(c) \cdot s_j$, with $s = \bigoplus_{j=1}^{d} s_j$. Therefore we have to combine leakages from each of the d computations to recover information about the secret s.

First, we can see that if there is no noise, it is still easy to recover the key using the attack of Section 4.2. If we send the challenge $c = 2^{79}$, we have $m_i(c) = X^{i-1}$ and $m_{i+1}(c) = X^i = m_i(c) \cdot X$. By comparing $\mathrm{HW}(X^{i-1} \cdot s_j)$ and

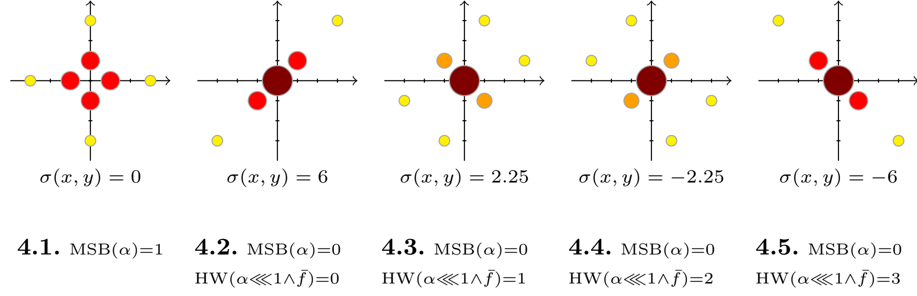

Fig. 4. Possible distributions for $\big(\mathrm{HW}(\alpha_1 \cdot X) - \mathrm{HW}(\alpha_1), \mathrm{HW}(\alpha_2 \cdot X) - \mathrm{HW}(\alpha_2)\big)$. The probabilities are represented as: yellow: 1/16, orange: 2/16, red: 3/16, dark red: 8/16

$\mathrm{HW}(X^i \cdot s_j)$, we can recover $\mathrm{MSB}(X^{i-1} \cdot s_j)$, and we can rebuild a bit of s using $\mathrm{MSB}(X^{i-1} \cdot s) = \bigoplus_j \mathrm{MSB}(X^{i-1} \cdot s_j)$.

More generally, we study the $2d$-dimensional distribution of:

$$\big(\mathrm{HW}(\alpha_j), \mathrm{HW}(\alpha_j \cdot X)\big)_{j=1}^{d}, \quad \text{with } \alpha = \textstyle\bigoplus_{j=1}^{d} \alpha_j.$$

Following the analysis of Section 4.2, we combine the measures using a difference operation, and reduce them to d dimensions:

$$\big(\mathrm{HW}(\alpha_j \cdot X) - \mathrm{HW}(\alpha_j)\big)_{j=1}^{d}, \quad \text{with } \alpha = \textstyle\bigoplus_{j=1}^{d} \alpha_j.$$

We will later use this analysis with $\alpha = m \cdot s$ and $\alpha_j = m \cdot s_j$.

We now study the case $d = 2$ in more details. Following the analysis of Section 4.2, we expect different distributions depending on the most significant bit of α.

MSB(α) = 1: If $\mathrm{MSB}(\alpha) = 1$, then we have either $\mathrm{MSB}(\alpha_1) = 0$ and $\mathrm{MSB}(\alpha_2) = 1$, or $\mathrm{MSB}(\alpha_1) = 1$ and $\mathrm{MSB}(\alpha_2) = 0$. This results in the distribution of Figure 4.1: either $\mathrm{HW}(\alpha_1 \cdot X) - \mathrm{HW}(\alpha_1) = 0$ and $\mathrm{HW}(\alpha_2 \cdot X) - \mathrm{HW}(\alpha_2) \in \{-3, -1, +1, +3\}$, or $\mathrm{HW}(\alpha_1 \cdot X) - \mathrm{HW}(\alpha_1) \in \{-3, -1, +1, +3\}$ and $\mathrm{HW}(\alpha_2 \cdot X) - \mathrm{HW}(\alpha_2) = 0$.

MSB(α) = 0: If $\mathrm{MSB}(\alpha) = 0$, then we have either $\mathrm{MSB}(\alpha_1) = 0$ and $\mathrm{MSB}(\alpha_2) = 0$, or $\mathrm{MSB}(\alpha_1) = 1$ and $\mathrm{MSB}(\alpha_2) = 1$. The first case gives $\mathrm{HW}(\alpha_1 \cdot X) - \mathrm{HW}(\alpha_1) = 0$ and $\mathrm{HW}(\alpha_2 \cdot X) - \mathrm{HW}(\alpha_2) = 0$. In the second case, we have $\mathrm{HW}(\alpha_1 \cdot X) - \mathrm{HW}(\alpha_1) \in \{-3, -1, +1, +3\}$ and $\mathrm{HW}(\alpha_2 \cdot X) - \mathrm{HW}(\alpha_2) \in \{-3, -1, +1, +3\}$, but we need to look at $\mathrm{HW}(\alpha \lll 1 \wedge \bar{f})$ in order to predict all the possibilities. The results are shown in Figure 4.

We can use those distributions to mount a template attack against a masked implementation of Lapin. We use $c = 2^{79}$, in order to collect traces with $\mathrm{HW}(X^{i-1} \cdot s_j)$ and $\mathrm{HW}(X^i \cdot s_j)$, and we recover $\mathrm{MSB}(X^{i-1} \cdot s)$ by distinguishing the distributions (*i.e.* we have $\alpha = X^{i-1} \cdot s$). We report our simulations results in Figure 7

in Appendix A. We can see that the data complexity increases roughly like the squared variance σ^4, which is typical of a second-order attack [5]. In our simulations, the rank of the correct key becomes smaller than 2^{10} when the data complexity is about $2^{10} \cdot \sigma^4$.

Order of the attack. This attack exploits information from four different measures, and combines pairs of measures using the difference operation. Then we have to distinguish the distributions of Figure 4, which can be done by computing the covariance and testing whether it is zero or in $\{-6, -2.25, 2.25, 6\}$. Therefore, this attack can be seen as a second-order 4-variate attack.

5 Conclusion

The previous results suggest that Lapin has interesting properties for secure and efficient masking, because it can be implemented by manipulating shares independently. They also exhibit that the exploitation of its leakage does not directly derive from standard DPA such as applied in the context of block ciphers. Yet, it is possible to mount attacks against both unprotected and masked Lapin, with similar intuition regarding the security order as for block ciphers. Admittedly, our side-channel experiments are only a first step, and several problems remain open. Technically, it would certainly be worth investigating other leakage models (*e.g.* distance-based) and actual measurements. Besides, it could be interesting to further study the possible presence of more data-dependent algorithmic noise in an implementation (*i.e.* capturing more than the main register activity), and how to get rid of it taking advantage of multiple plaintexts in a collision-like attack. Eventually, and as pointed out in introduction, the problem of on-chip randomness generation remains an important drawback of Lapin. Analyzing its leakage, or designing deterministic protocols based on the Learning With Rounding assumption are interesting scopes for further research.

References

1. Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with Rounding, Revisited - New Reduction, Properties and Applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 57–74. Springer, 2013.
2. Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom Functions and Lattices. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 719–737. Springer, 2012.
3. Daniel J. Bernstein and Tanja Lange. Never Trust a Bunny. In Jaap-Henk Hoepman and Ingrid Verbauwhede, editors, *RFIDSec*, volume 7739 of *Lecture Notes in Computer Science*, pages 137–148. Springer, 2012.
4. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Joye and Quisquater [10], pages 16–29.

5. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
6. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.
7. Louis Goubin and Jacques Patarin. DES and Differential Power Analysis (The "Duplication" Method). In Çetin Kaya Koç and Christof Paar, editors, *CHES*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.
8. Vincent Grosso, François-Xavier Standaert, and Sebastian Faust. Masking vs. Multiparty Computation: How Large Is the Gap for AES? In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 400–416. Springer, 2013.
9. Stefan Heyse, Eike Kiltz, Vadim Lyubashevsky, Christof Paar, and Krzysztof Pietrzak. Lapin: An Efficient Authentication Protocol Based on Ring-LPN. In Anne Canteaut, editor, *FSE*, volume 7549 of *Lecture Notes in Computer Science*, pages 346–365. Springer, 2012.
10. Marc Joye and Jean-Jacques Quisquater, editors. *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*. Springer, 2004.
11. Stefan Mangard. Hardware Countermeasures against DPA ? A Statistical Analysis of Their Effectiveness. In Tatsuaki Okamoto, editor, *CT-RSA*, volume 2964 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 2004.
12. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.
13. Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In Alfred Menezes, editor, *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.
14. Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully Attacking Masked AES Hardware Implementations. In Josyula R. Rao and Berk Sunar, editors, *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2005.
15. Marcel Medwed and François-Xavier Standaert. Extractors against side-channel attacks: weak or strong? *J. Cryptographic Engineering*, 1(3):231–241, 2011.
16. Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.
17. Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar. A Collision-Attack on AES: Combining Side Channel- and Differential-Attack. In Joye and Quisquater [10], pages 163–175.
18. François-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. The World Is Not Enough: Another Look on Second-Order DPA. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2010.
19. Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Security Evaluations beyond Computing Power. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2013.

A Simulation results of the side-channel attacks

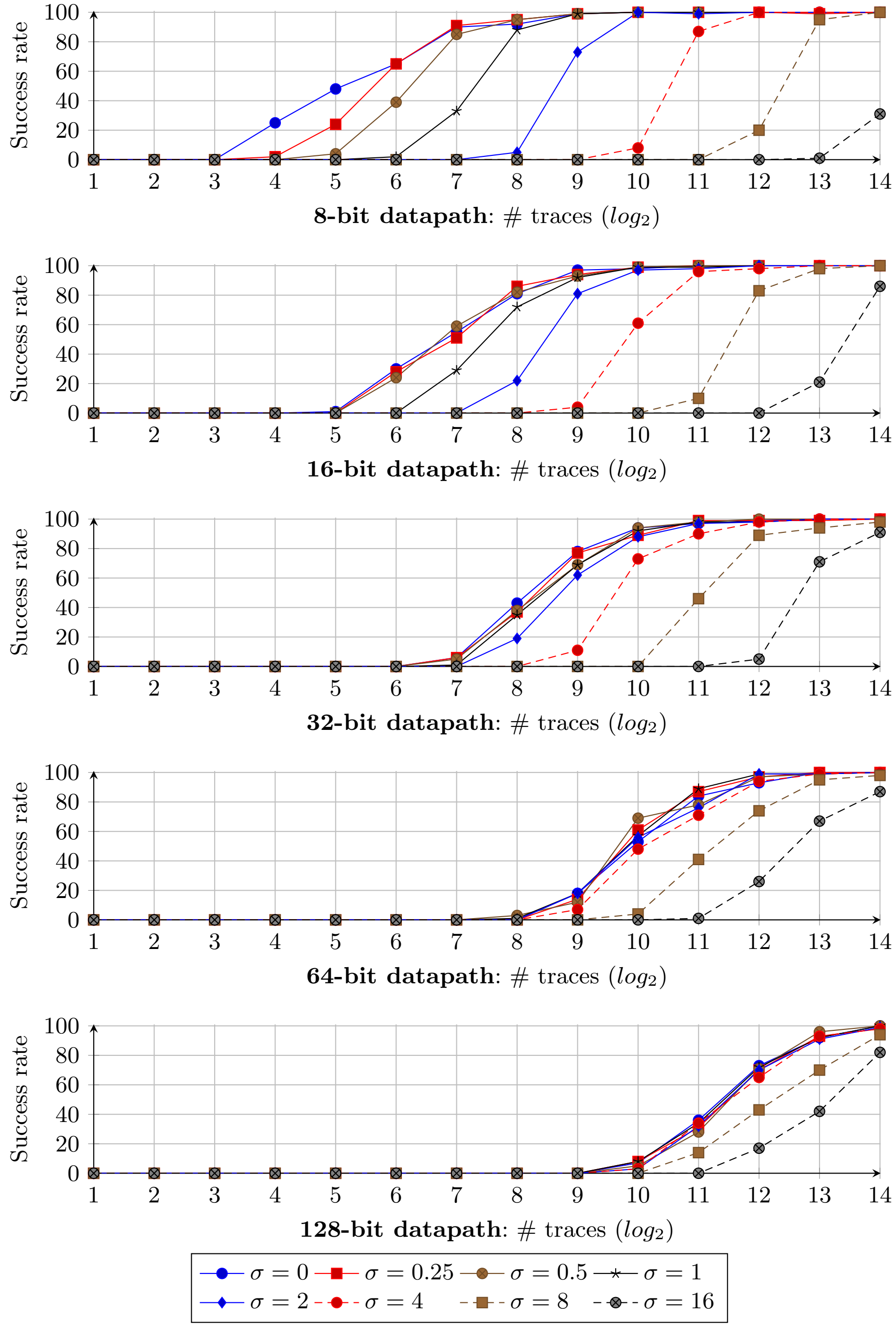

Fig. 5. DPA attack success rate for full-key recovery, after exploring 2^{16} tree nodes.

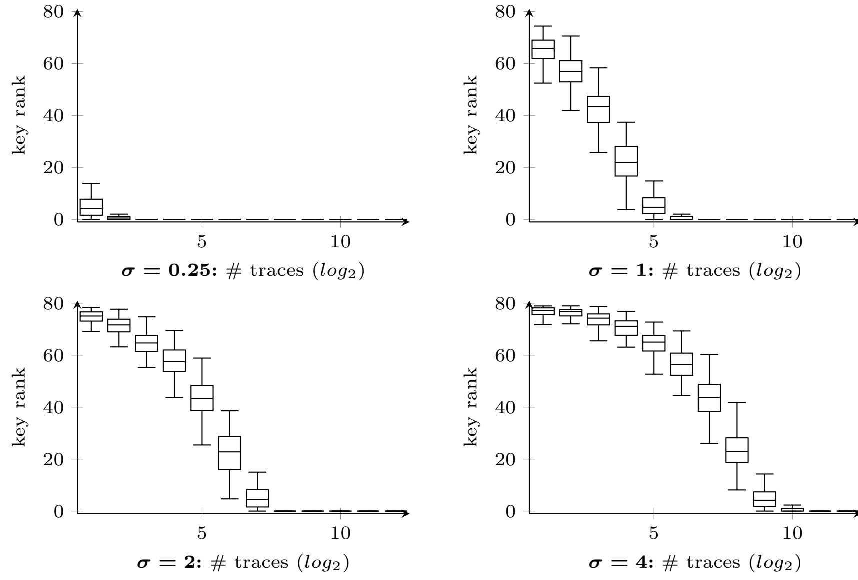

Fig. 6. Security graphs for the collision-like attack, with $k = 128$. We assume that all the clock cycles are used for each trace. Alternatively, the attack can be mounted targeting a single point of interest if the data complexity is multiplied by 80.

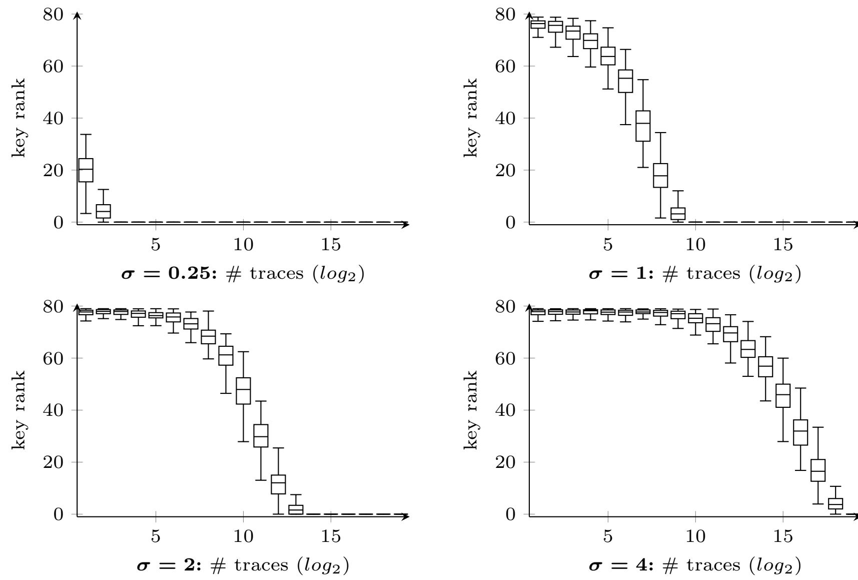

Fig. 7. Security graphs for the collision-like attack on a masked Lapin, with $k = 128$.

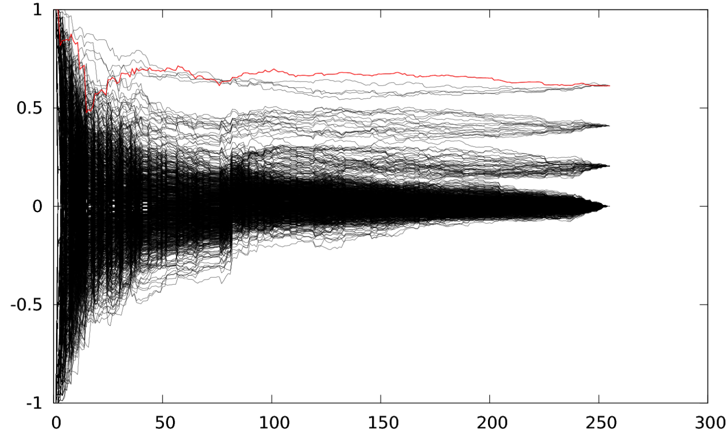

Fig. 8. Correlation coefficient for the key candidates, depending on the number of traces. We use $t = 7$ and $p = 3$, and don't add any noise to the Hamming weights.

B Additionnal implementation results

Table 2. Number of clock cycles required for Lapin calculation.

	One share ($d = 1$)					Three shares ($d = 3$)				
	8-bit	16-bit	32-bit	64-bit	128-bit	8-bit	16-bit	32-bit	64-bit	128-bit
f_1	4,048	2,024	1,012	506	257	12,144	6,072	3,036	1,518	771
f_2	4,160	2,080	1,040	520	264	12,480	6,240	3,120	1,560	792
f_3	4,208	2,104	1,052	526	267	12,624	6,312	3,156	1,578	801
f_4	4,224	2,112	1,056	528	268	12,672	6,336	3,168	1,584	804
f_5	4,336	2,168	1,084	542	275	13,008	6,504	3,252	1,626	825
Total	20,977	10,489	5,245	2,623	1,332	62,961	31,481	15,741	7,871	3,996