Side-channel countermeasures utilizing dynamic logic reconfiguration: protecting AES/Rijndael and Serpent encryption in hardware

Petr, S.; Brejník, J.; Balasch, J.; Novotný, M.; Mentens, N.

Microprocessors and Microsystems 78 (2020) 103208

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Side-channel countermeasures utilizing dynamic logic reconfiguration: Protecting AES/Rijndael and Serpent encryption in hardware

Petr Socha [a,*], Jan Brejník [a], Josep Balasch [b,c], Martin Novotný [a], Nele Mentens [d,c,e]

[a] Czech Technical University in Prague, Faculty of Information Technology, Czech Republic
[b] KU Leuven, eMedia lab, Belgium
[c] KU Leuven, imec-COSIC/ESAT, Belgium
[d] KU Leuven, ES&S, Belgium
[e] Leiden University, LIACS, The Netherlands

ARTICLE INFO

Keywords:
Internet of things
Embedded security
Cryptography
Side-channel analysis
Dynamic logic reconfiguration

ABSTRACT

Dynamic logic reconfiguration is a concept that allows for efficient on-the-fly modifications of combinational circuit behavior in both ASIC and FPGA devices. The reconfiguration of Boolean functions is achieved by modification of their generators (e.g., shift register-based look-up tables) and it can be controlled from within the chip, without the necessity of any external intervention. This hardware polymorphism can be utilized for the implementation of side-channel attack countermeasures, as demonstrated by Sasdrich et al. for the lightweight cipher PRESENT.

In this work, we adapt these countermeasures to two of the AES finalists, namely Rijndael and Serpent. Just like PRESENT, both Rijndael and Serpent are block ciphers based on a substitution–permutation network. We describe the countermeasures and adjustments necessary to protect these ciphers using the resources available in modern Xilinx FPGAs. We describe our implementations and evaluate the side-channel leakage and effectiveness of different countermeasures combinations using a methodology based on Welch's t-test. Furthermore, we attempt to break the protected AES/Rijndael implementation using second-order DPA/CPA attacks.

We did not detect any significant first-order leakage from the fully protected versions of our implementations. Using one million power traces, we detect second-order leakage from Serpent encryption, while AES encryption second-order leakage is barely detectable. We show that the countermeasures proposed by Sasdrich et al. are, with some modifications, successfully applicable to AES and Serpent.

1. Introduction

The use of computers and various embedded systems has become our daily routine in the past years. In the upcoming Internet-of-Things (IoT) era, smart cities and smart homes are expected to bring even more embedded devices into our everyday lives. The presence of such smart devices, including personal assistants, cars, and many more, makes our private lives more vulnerable than ever before. In order to protect sensitive information, various authentication, authorization, and encryption schemes need to be employed. Even though these algorithms may be considered secure, their implementations may still be vulnerable to side-channel attacks. These attacks exploit the fact that sensitive information may leak through side channels, such as the power consumption of the device [1,2] or its electromagnetic radiation [3]. Given the typical deployment of IoT devices, where the attacker may easily gain physical access and tamper with the device, these attacks pose a severe threat.

Many different countermeasures have been proposed to prevent side-channel attacks. Masking is a popular approach based on randomizing intermediate cipher values by introducing a random mask [4,5], making it difficult for an attacker to predict the processed values. Another approach, called hiding, tries to hide the information leakage, e.g. through the use of dual-rail logic [6]. Dynamic reconfiguration has been proposed as another hiding countermeasure to achieve side-channel resistance [7]. A combination of countermeasures implemented using dynamic logic reconfiguration is proposed in [8] and evaluated on the lightweight block cipher PRESENT [9].

In this paper, we extend the work presented in [8] by using dynamic logic reconfiguration to secure two of the Advanced Encryption

* Corresponding author.
E-mail addresses: petr.socha@fit.cvut.cz (P. Socha), brejnjan@fit.cvut.cz (J. Brejník), josep.balasch@kuleuven.be (J. Balasch), novotnym@fit.cvut.cz (M. Novotný), nele.mentens@kuleuven.be (N. Mentens).

https://doi.org/10.1016/j.micpro.2020.103208
Received 11 December 2019; Received in revised form 20 July 2020; Accepted 21 July 2020
Available online 3 August 2020

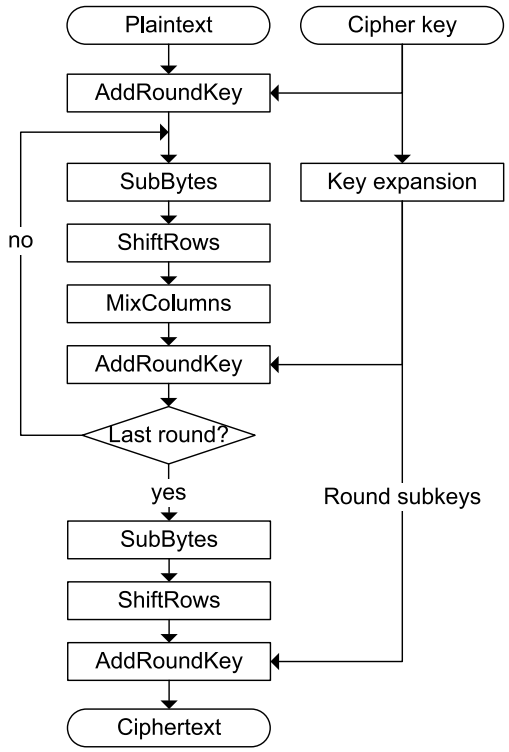


Fig. 1. AES/Rijndael encryption.

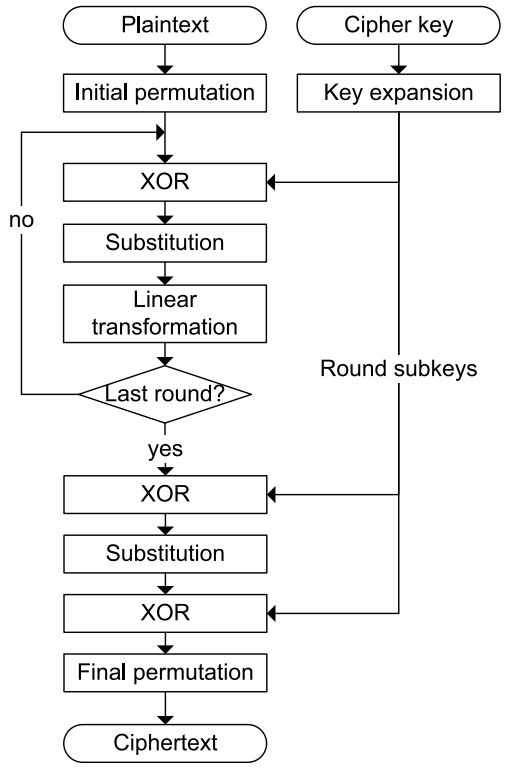


Fig. 2. Serpent encryption.

Standard (AES) competition finalists, Rijndael [10] (winner of the competition, nowadays therefore known as the AES) and Serpent [11]. We describe our implementations and the non-straightforward way in which we tailored the countermeasures in [8] to AES and Serpent. We evaluate the side-channel leakage and the effectiveness of different countermeasures combinations.

2. Theoretical background

In this work, we intend to secure AES and Serpent using the approach described in [8]. In the following subsections, we first describe both AES/Rijndael and Serpent. Then we explain the concept of dynamic logic reconfiguration on FPGA, and finally, we describe the implemented and evaluated countermeasures.

2.1. AES finalists: Rijndael and Serpent

Both ciphers share common features [12]. They are iterated substitution–permutation networks (SPN) with a block size of 128 bits and possible key sizes of 128, 192, or 256 bits. The plaintext (i.e. the data to be encrypted) is transformed into a ciphertext by iteratively applying a number of operations. Each iteration is called a round. Both ciphers also describe a method for expanding the secret key into a number of subkeys, which are used as an input to each round.

2.1.1. AES/Rijndael

Rijndael [10] consists of 10, 12, or 14 rounds (depending on the key length). First, the secret key is XORed with the plaintext. After that, a number of round transformations is performed. Each round consists of four layers: a non-linear substitution layer (SubBytes, i.e., 16 parallel applications of an 8-bit substitution box or S-box), two linear mixing layers (ShiftRows and MixColumns) and an XOR with the round subkey (AddRoundKey). In the last round, the MixColumns transformation is omitted. The Rijndael encryption is depicted in Fig. 1.

2.1.2. Serpent

Serpent [11] consists of 32 rounds. First, an initial permutation is applied, and then the round transformations take place. Each round consists of three layers: an XOR with the round subkey, a non-linear substitution layer (i.e., 32 parallel applications of one of the eight specified 4-bit S-boxes, which are different in the consecutive rounds), and a linear transformation. In the last round, a second XOR takes place instead of the linear transformation. In the end, the final permutation is applied. The Serpent encryption is depicted in Fig. 2.

2.2. Dynamic logic reconfiguration

Dynamic logic reconfiguration is a concept that allows for efficient on-the-fly modifications of combinational circuit behavior in both ASIC [13,14] and FPGA devices. In FPGAs, combinational circuits are typically implemented using Look-Up Tables (LUTs), i.e., configurable primitives which store truth tables of k-input Boolean functions $f : \mathbb{B}^k \rightarrow \mathbb{B}$. Dynamic logic reconfiguration allows for the run-time alteration of the circuit behavior by modifying the content of specific look-up tables, while leaving the routing intact. The reconfiguration of LUTs is done from within the chip itself and can be achieved, e.g., by using a shift register (allowing for serial programming) and a cascade of addressing multiplexers. This concept is demonstrated in Fig. 3.

In Xilinx FPGAs [15], this functionality is provided by k-input Configurable Look-Up Tables (CFGLUTs) with a serial configuration input and output (allowing to connect CFGLUTs in separate configuration chains). In Xilinx Spartan-6 FPGAs, 5-input CFGLUTs are available.

In order to implement dynamically reconfigurable Boolean functions $f : \mathbb{B}^n \rightarrow \mathbb{B}$, where $n > k$, multiple k-input CFGLUTs are required in combination with addressing multiplexers (using Boole's expansion, also referred to as the Shannon expansion [16]). Specifically, to implement an n-input function using k-input CFGLUTs and 2-to-1 multiplexers, we need 2^{n-k} CFGLUTs and $2^{n-k} - 1$ multiplexers.

Multiple-output Boolean functions $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ can be trivially implemented as m single-output Boolean functions $f_i : \mathbb{B}^n \rightarrow \mathbb{B}$.

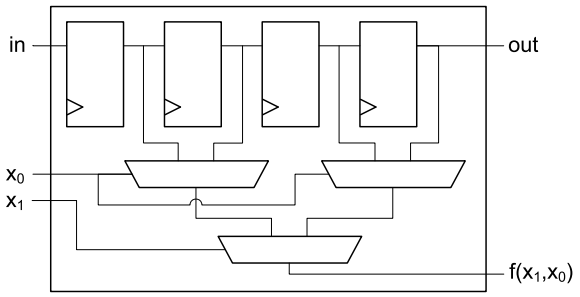
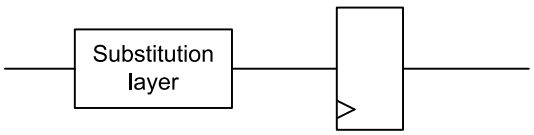
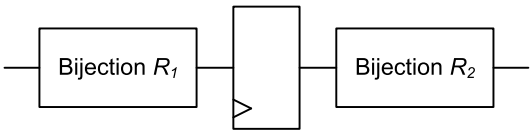


Fig. 3. Example of a 2-input reconfigurable look-up table with serial programming I/O.



(a) Unprotected substitution layer.



(b) Substitution layer decomposed into two bijections.

Fig. 4. S-box Decomposition.

2.3. Countermeasures

To protect AES and Serpent, we have implemented countermeasures proposed (and evaluated on PRESENT) by Sasdrich et al. in [8]. In this subsection, we briefly describe these countermeasures.

2.3.1. S-box decomposition

Since information leakage often occurs based on changing values in registers, and since the output of the non-linear substitution layer is a frequent target of side-channel attacks, the S-box decomposition countermeasure is based on avoiding the storage of the S-box outputs into such registers. This is done by decomposing the S-box into two bijections R_1, R_2, where

$$\text{S-box}(x) = R_2(R_1(x)), \tag{1}$$

and by placing the register in between the two bijections. The decomposition is demonstrated in Fig. 4. The number of possible n-bit bijections for R_1 is equal to $(2^n)!$. For each option, a bijection R_2 can be found such that Eq. (1) holds.

Thanks to dynamic logic reconfiguration, different bijections R_1, R_2 can easily be used for every encryption. Starting with R_1 being an identity and R_2 being the actual S-box (or vice versa), the bijections for the next encryption are computed by randomly selecting pair(s) of elements in the R_1 mapping, swapping them, and recomputing R_2 accordingly.

2.3.2. Boolean masking

In order to randomize intermediate values, a random mask is added (XORed) to the data prior to encryption, and subtracted (i.e. once again XORed) after the encryption. For the cipher to produce valid results working with masked data, various alterations must be done.

Boolean masking can be combined with the previously mentioned bijective S-box decomposition and can once again take advantage of

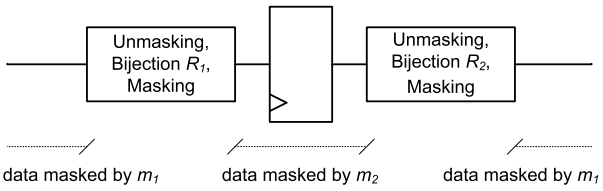


Fig. 5. S-box Decomposition + Masking: all the three operations (unmasking, bijection and masking) are performed as a single table lookup, therefore unmasked data does not appear on any wires at any time.

dynamic logic reconfiguration. Two different random masks m_1, m_2 are used for every encryption: mask m_1 is used outside the decomposed S-box, and mask m_2 is used inside of it. If the substitution layer were the only layer in the round, the previously mentioned bijections R_1, R_2 would get adjusted as follows:

$$R_1'(x) = R_1(x \oplus m_1) \oplus m_2, \tag{2}$$

$$R_2'(x) = R_2(x \oplus m_2) \oplus m_1. \tag{3}$$

The function R_1' first subtracts/removes mask m_1, then performs the R_1 bijection mapping, and finally masks this value using m_2. The output of this function is stored in the register. Analogically, the function R_2' subtracts the mask m_2, does the R_2 mapping, and masks the result using m_1. This is demonstrated in Fig. 5. This way, the same CFGLUTs can be used for both the S-box decomposition and the masking, saving both area and reconfiguration time.

However, to deal with the linear transformation layers, further alterations to the R_1', R_2' bijections need to be done. We can exploit one of these two facts:

$$f(x) = f(x \oplus f^{-1}(m)) \oplus m, \tag{4}$$

$$f(x) = f(x \oplus m) \oplus f(m), \tag{5}$$

which both hold when f is a linear mapping. These give us two different and fairly straightforward approaches to take linear transformations f into account.

One option is to alter R_2' function in terms of Eq. (4) so that m_1 processed by the inverse transformation is used to mask the data, allowing to subtract m_1 in R_1':

$$R_1'(x) = R_1(x \oplus m_1) \oplus m_2, \tag{6}$$

$$R_2'(x) = R_2(x \oplus m_2) \oplus f^{-1}(m_1). \tag{7}$$

The second option is to use m_1 for masking in R_2', and to alter R_1' according to Eq. (5), so that m_1 processed by the linear transformation gets subtracted:

$$R_1'(x) = R_1(x \oplus f(m_1)) \oplus m_2, \tag{8}$$

$$R_2'(x) = R_2(x \oplus m_2) \oplus m_1. \tag{9}$$

Notice that further alterations may be required for the first and the last round, depending on the selected approach.

The last obstacle is the subkey XOR layer, which can be considered an affine transformation. Suppose we have a vector x, which gets XORed with the subkey: $x \oplus k$. Suppose we process masked data the same way: $(x \oplus m) \oplus k$, then by subtracting the mask m with no alterations we have:

$$(((x \oplus m) \oplus k) \oplus m) = x \oplus k. \tag{10}$$

Therefore, no further alterations need to be done to take the XOR layer into account.

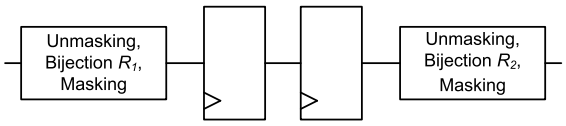


Fig. 6. S-box Decomposition + Masking + Register Precharge.

2.3.3. Register precharge

Because the same masks are used for the whole encryption (i.e., for every round), the leakage occurs in the register, since

$$\mathrm{HD}(x \oplus m, y \oplus m) = \mathrm{HD}(x, y), \tag{11}$$

where $\mathrm{HD}(x, y)$ denotes the Hamming distance between x and y. To avoid this leakage, the register is duplicated, as shown in Fig. 6, and the processed data are interleaved with random data. This technique avoids leakage; however, it reduces the throughput of the circuit when it is implemented using an architecture that is not fully unrolled.

3. Secure cipher design

In this section, we examine the specifics of both AES/Rijndael and Serpent and we propose a manner in which these ciphers can be secured against side-channel attacks using the countermeasures explained in Section 2.3.

In order for our implementations to fit into a Xilinx Spartan-6 FPGA device, we take into account that CFGLUTs with at most 5 input bits are available. When a platform with smaller CFGLUTs is available, the dynamic logic reconfiguration method can be implemented using the approach described in Section 2.2.

3.1. AES/Rijndael

Rijndael employs an 8×8 S-box, which can be considered as a function $\text{S-box}_{\text{Rijndael}} : \mathbb{B}^8 \rightarrow \mathbb{B}^8$. Therefore, to implement the Rijndael S-box using reconfigurable logic, $8 \cdot 2^{8-5} = 64$ (5-input) CFGLUTs and $8 \cdot (2^{8-5} - 1) = 56$ (2-to-1) multiplexers are necessary. Moreover, the S-box decomposition countermeasure suggests the S-box to be split into two bijections $R_1, R_2 : \mathbb{B}^8 \rightarrow \mathbb{B}^8$, which doubles the amount of CFGLUTs and multiplexers in the secured version. Since the Rijndael algorithm applies 16 S-boxes in parallel, this brings the total count up to 2048 (5-input) CFGLUTs and 1792 (2-to-1) multiplexers.

The decomposition into two bijections is done in a similar fashion as described in Section 2.3, with the round register being placed in between the two bijections. For the AES algorithm, we have decided to swap eight pairs of elements in the R_1 bijection after every encryption (in contrast to the PRESENT 4-bit S-box decomposition in [8], where only a single pair gets swapped).

To implement the Boolean masking countermeasure as described in Section 2.3, bijections R'_1, R'_2 (i.e. the decomposed S-box combined with masking) must be altered. We choose the option where R'_2 adds the mask m_1 and R'_1 subtracts m_1 processed by the linear transformations (see Eq. (8)):

$$R'_1(x) = R_1(x \oplus \mathrm{MixColumns}(\mathrm{ShiftRows}(m_1))) \oplus m_2, \tag{12}$$

$$R'_2(x) = R_2(x \oplus m_2) \oplus m_1. \tag{13}$$

Note that the data are masked by m_1 in the second bijection R_2 and that this mask is subtracted in the following round. Therefore prior to the first round, the input data must be masked properly. Also, the last round of Rijndael omits the MixColumns operation, so additional unmasking of the output must be done with this in mind.

The implementation of the register precharge requires the register to be duplicated and the controller to be adjusted appropriately, such that the processed data are interleaved with random data.

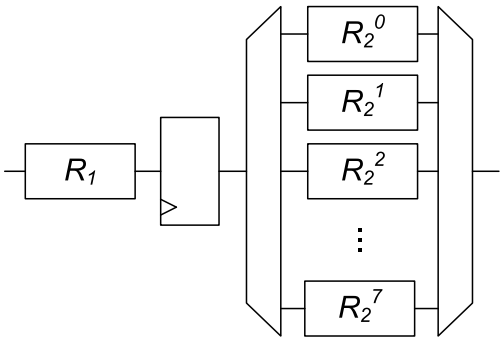


Fig. 7. Serpent S-boxes decomposition; notice the demultiplexer, which is necessary to prevent glitches.

3.2. Serpent

Unlike Rijndael or PRESENT, Serpent defines eight different 4×4 S-boxes. Each S-box is used in a different round. One way to implement the S-box decomposition is to decompose each of these S-boxes into two bijections, resulting in 16 bijections in total. We have decided for an approach where the first bijection R_1 is shared among all S-boxes, while the other eight bijections R_2^i, $i \in \{0, \ldots, 7\}$, implement the eight S-boxes, with the correct output being selected by a multiplexer. The eight decomposed Serpent S-boxes are depicted in Fig. 7. Notice the **demultiplexer**, which selects the right R_2^i bijection, while the other bijections are fed with zeros. This demultiplexer is necessary to prevent glitches that lead to information leakage. Since the Serpent S-boxes implement the functions $\text{S-box}^{\text{i}}_{\text{Serpent}} : \mathbb{B}^4 \rightarrow \mathbb{B}^4$, only four CFGLUTs are necessary to implement the bijection. Given the selected architecture, $4+8 \cdot 4 = 36$ CFGLUTs are required to decompose all eight S-boxes. Since the S-box is applied 32 times in parallel, this results in 1152 CFGLUTs in total.

Boolean masking is implemented similarly to the Rijndael algorithm, with m_1, processed by the linear transformation, being subtracted in the R'_1 bijection (see Eq. (8)). Suppose the Serpent linear transformation is $L_{Serpent}$, then:

$$R'_1(x) = R_1(x \oplus \mathrm{L}_{\mathrm{Serpent}}(m_1)) \oplus m_2, \tag{14}$$

$$R'_2(x) = R_2(x \oplus m_2) \oplus m_1. \tag{15}$$

Regarding the first round, similarly to the Rijndael approach, appropriate initial masking of the input data must be performed first. Also, there is no linear transformation in the last round; therefore the unprocessed mask m_1 gets subtracted during the final unmasking.

Register precharge is once again implemented simply by duplicating the round register and altering the controller appropriately to interleave the processed data with random data.

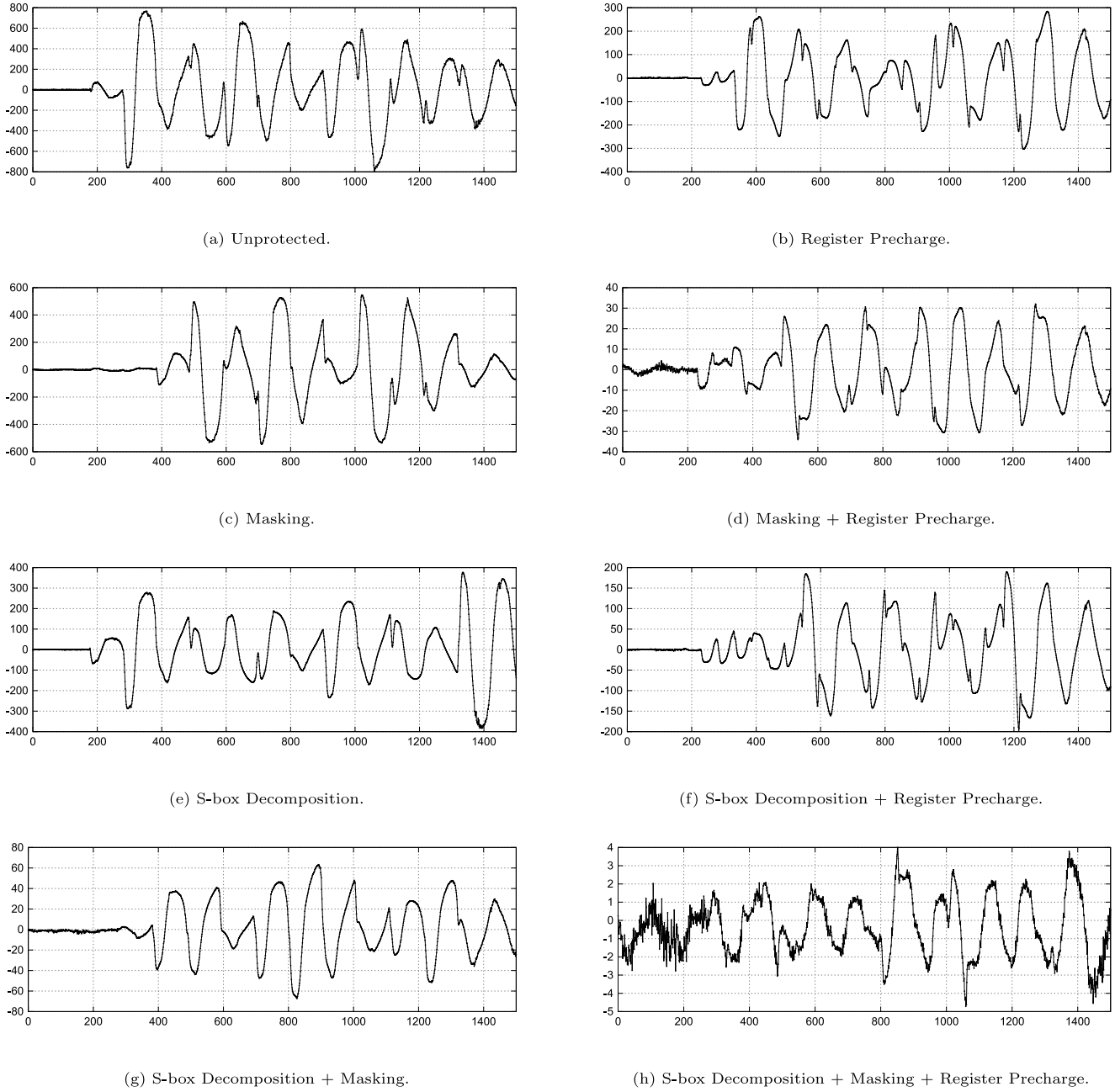
3.3. Latency and area utilization

For every encryption, new bijections are generated (as described in Section 2.3), as well as new masks m_1, m_2. This requires the CFGLUTs configurations to be computed and loaded prior to every encryption. The reconfiguration of all CFGLUTs can be done using different levels of parallelism (the CFGLUTs "programming" I/O can be variously chained, given its shift register nature). The serial reconfiguration of n-input CFGLUT requires 2^n cycles, therefore selected reconfiguration strategy has a direct impact on the overall latency, as well as on the area utilization.

Table 1 presents a comparison of the latency and the area of both unprotected and protected AES/Rijndael and Serpent encryption im-

Table 1
Latency and area utilization.

Implementation	Area		Latency (clock cycles)		
	Memory (FFs)	Logic (LUTs)	Encryption	Extra	Total
Unprotected AES/Rijndael, LUT-based S-Boxes	278	1,304	10	0	10
Protected AES/Rijndael, 2 CFGLUT chains	1,073	2,652	20	4,122	4,142
Protected AES/Rijndael, 32 CFGLUT chains	3,229	7,234	20	282	302
Unprotected Serpent, LUT-based S-Boxes	430	1,660	32	0	32
Protected Serpent, 9 CFGLUT chains	2,945	5,696	64	538	602
Protected Serpent, 144 CFGLUT chains	4,441	9,471	64	58	122
Protected Serpent, 288 CFGLUT chains	6,040	13,211	64	42	106



(a) Unprotected.

(b) Register Precharge.

(c) Masking.

(d) Masking + Register Precharge.

(e) S-box Decomposition.

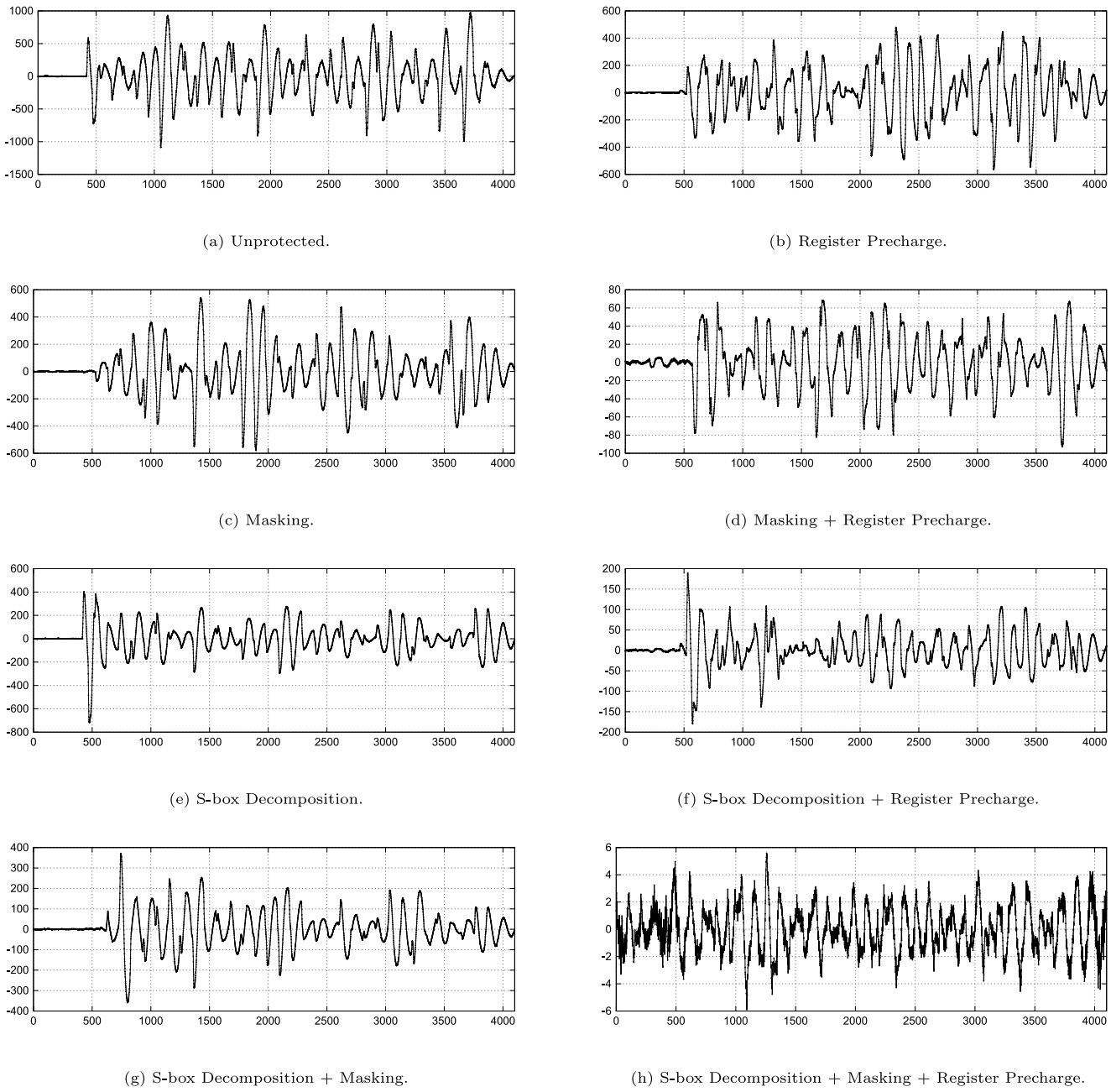
(f) S-box Decomposition + Register Precharge.

(g) S-box Decomposition + Masking.

(h) S-box Decomposition + Masking + Register Precharge.

Fig. 8. Results of the AES/Rijndael t-test, where the t-value is shown on the vertical axis and the time samples during encryption are shown on the horizontal axis.

plementations. The Flip-Flop (FF) and the Look-Up Table (LUT) counts are Xilinx ISE post-synthesis statistics for Xilinx Spartan-6 FPGA. The encryption latency of protected implementations is double due to the register precharge. The extra latency is caused mostly by the CFGLUT serial programming, and it can be reduced by using several parallel configuration chains, at the expense of the area.



(a) Unprotected.

(b) Register Precharge.

(c) Masking.

(d) Masking + Register Precharge.

(e) S-box Decomposition.

(f) S-box Decomposition + Register Precharge.

(g) S-box Decomposition + Masking.

(h) S-box Decomposition + Masking + Register Precharge.

Fig. 9. Results of the Serpent t-test, where the t-value is shown on the vertical axis and the time samples during encryption are shown on the horizontal axis.

4. Side-channel leakage evaluation

In this section, we present our experimental setup and a leakage methodology used to evaluate all combinations of previously described countermeasures.

4.1. Measurement setup

We choose the Sakura-G board [17] with a Xilinx Spartan-6 FPGA as our evaluation platform. AES/Rijndael and Serpent VHDL implementations with a 128-bit key are evaluated. The power traces evaluated in Sections 4.2, 4.3 and 4.5 are measured using a PicoScope 6406D oscilloscope and the power traces evaluated in Section 4.4 are measured using a Textronix DPO 7254 oscilloscope. The current consumption of the FPGA core is measured as a voltage drop across a shunt resistor in the VCCINT path of the FPGA. The voltage drop is furthermore amplified using a built-in preamplifier before sampling by the oscilloscope. The sample rate used for all the measurements is 625 MS/sec.

4.2. First-order test vector leakage assessment

Leakage is evaluated using the non-specific univariate first-order Welch's t-test, as described in [18]. This evaluation method consists of two phases. In the active phase, power traces are collected, each trace measured while encrypting either a random or a (preselected) constant plaintext, resulting in two sets of power traces. In the analytical phase of the evaluation, Welch's t-test statistic is computed independently at each time sample:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}, \tag{16}$$

where $\bar{X}_1$, $\bar{X}_2$ are sample means, s_1^2, s_2^2 are sample variances, and N_1, N_2 are sample sizes of the two sets of power traces at a given time sample, respectively. The Welch's t-test statistic examines the null hypothesis of equal population means (where one population consists of random plaintext measurements and the other population consists of constant plaintext measurements). In our case, the null hypothesis can be formulated in the sense that the two populations are not distinguishable by their sample means, which means that the sample means are not data-dependent. The null hypothesis gets rejected for high values of $|t|$, the threshold is usually set around 4.5 or 5.

4.2.1. Results

The necessary random data (random pairs to be swapped in the bijection, random masks, register precharge with random values) are generated externally and sent to the cryptographic device alongside the plaintext. This approach allows us to enable or disable specific countermeasures easily.

We evaluate every possible combination of the proposed countermeasures:

(a) Unprotected
(b) Register Precharge
(c) Masking
(d) Masking + Register Precharge
(e) S-box Decomposition
(f) S-box Decomposition + Register Precharge
(g) S-box Decomposition + Masking
(h) S-box Decomposition + Masking + Register Precharge

For every combination, one million power traces are measured and processed using a non-specific first-order t-test, as described earlier. Fig. 8 depicts the t-values during the AES encryption and Fig. 9 depicts the t-values during the Serpent encryption. The sensitive information leakage is the most prominent for the unprotected versions, as expected.

It is also visible that different countermeasures and their combinations have various influence on the significance of the detected leakage. Figs. 8(c) and 9(c) show that a countermeasure based on masking protects solely the first round of the cipher, while, starting from the second round, the leakage is comparable to the unprotected version (cf. Figs. 8(a) and 9(a)). Figs. 8(d) and 9(d) suggest that masking becomes more effective in combination with register precharge (which is expected, as discussed in Section 2.3).

Figs. 8(h) and 9(h) show results with all three countermeasures combined. As can be seen, no significant first-order leakage is detected when evaluating these fully protected implementations. However, the used Test Vector Leakage Assessment (TVLA) methodology is merely a first step in the evaluation of a side-channel security of the implementations and the results do not provide any guarantee of a security level [19]. This is not only because of a high risk of both false positives and false negatives, but also because only univariate statistics is considered in this methodology.

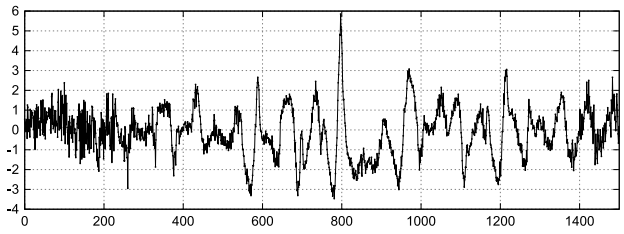
4.3. Second-order test vector leakage assessment

Protected implementations which make use of Boolean masking (i.e., splitting a working variable into d shares) are typically vulnerable against higher-order DPA attacks [20–22], i.e., attacks which exploit leakage from several variable shares, either by combining multiple time samples together (multivariate), or by analyzing higher statistical moments at a single time sample (univariate). Since a first-order masking scheme is used to protect our implementations, we assume them to be vulnerable against univariate second-order DPA.

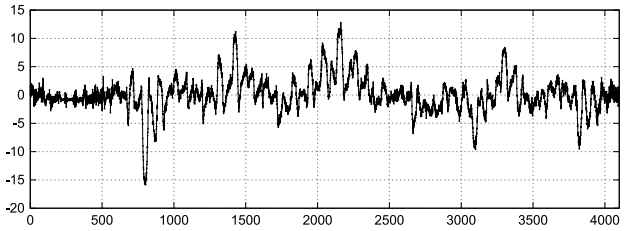
We evaluate the second-order leakage of our implementations using Welch's t-test, similar to the first-order leakage evaluation in Section 4.2. The first phase of the methodology stays the same —- therefore, we can use the same sets of power traces obtained for the first-order analysis. To analyze the second statistical moment, the power traces are preprocessed, at each time sample independently, by making every sample mean-free squared:

$$x' = (x - \bar{X})^2, \tag{17}$$

where $\bar{X}$ is sample mean at a given time sample. Then the Welch's t-test statistic is computed, same as in case of the first-order leakage evaluation.



(a) AES/Rijndael second-order t-test.



(b) Serpent second-order t-test.

Fig. 10. Results of the univariate second-order t-test with all the countermeasures enabled (S-box Decomposition + Masking + Register Precharge), where the t-value is shown on the vertical axis and the time samples during encryption are shown on the horizontal axis.

4.3.1. Results

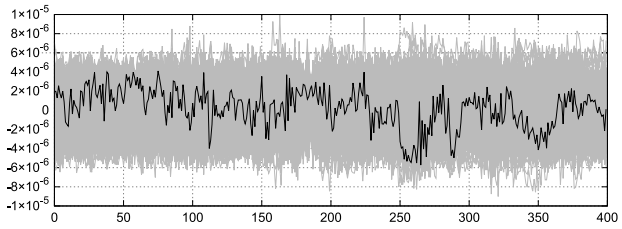
We evaluate one million previously captured power traces with all the proposed countermeasures enabled (S-box Decomposition + Masking + Register Precharge). Fig. 10 depicts (univariate) second-order t-values during AES and Serpent encryption. For AES, there is a single peak reaching as high as 6 halfway the encryption, as can be seen in Fig. 10(a). Second-order leakage is more prominent during Serpent encryption, as can be seen in Fig. 10(b), where the absolute t-value reaches as high as 15.

It is fair to assume that with more than one million power traces available, the second-order leakage would get more prominent and easier to detect. As shown in [21], the amount of power traces required to successfully mount a higher-order side-channel attack increases exponentially with the masking order.

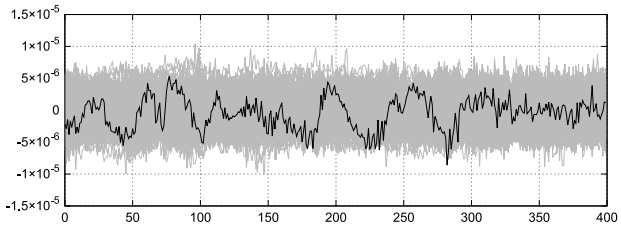
4.4. Second-order attacks

To provide more confidence about the AES/Rijndael implementation resilience, we attempt to break the fully protected implementation using second-order attacks [20]. The measured power traces are first preprocessed in the same fashion as in the case of the univariate second-order leakage assessment, as described in Section 4.3. Afterward, we perform the DPA attack [1] (using t-test distinguisher) and the CPA attack [2], targeting the first and the last cipher round, and considering both Hamming weight and Hamming distance leakage.

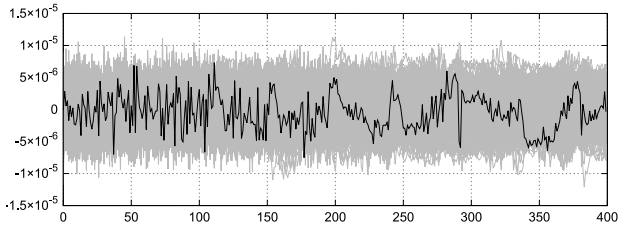
First, we target the AES S-box output after the first round, i.e., $s_i = \mathrm{S{-}box}(pt_i \oplus keyhypo_i)$ for some index $1 \leq i \leq 16$, key hypothesis $keyhypo_i \in [0, 255]$ and plaintext byte pt_i. Second, we target the AES



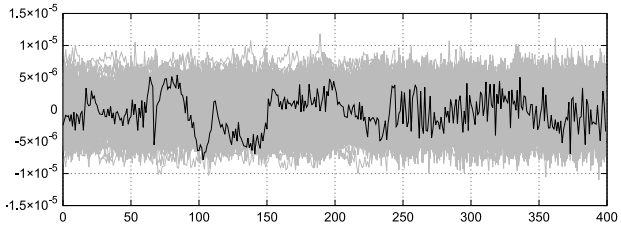
(a) AES/Rijndael, Hamming weight DPA (t-test) first round.



(b) AES/Rijndael, Hamming weight DPA (t-test) last round.

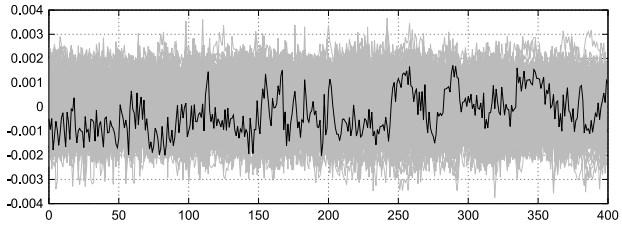


(c) AES/Rijndael, Hamming distance DPA (t-test) first round.

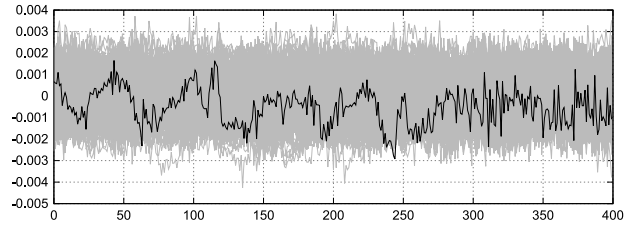


(d) AES/Rijndael, Hamming distance DPA (t-test) last round.

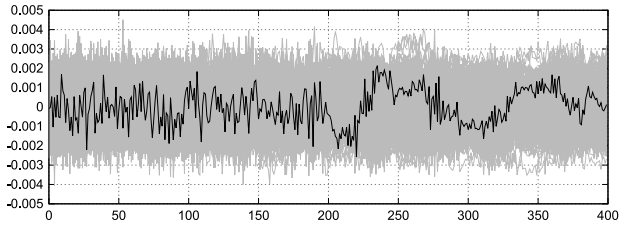
Fig. 11. Second-order DPA attack (using t-test distinguisher) on first and last round AES/Rijndael subkey (byte #1), where the t-value is shown on the vertical axis and the time samples during the relevant encryption rounds are shown on the horizontal axis.



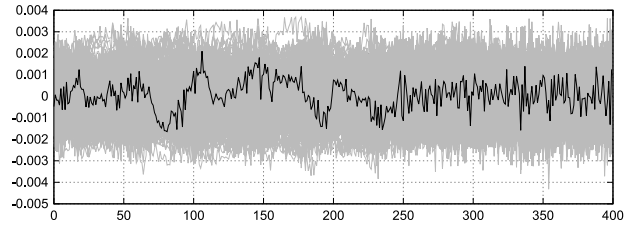
(a) AES/Rijndael, Hamming weight CPA (Pearson) first round.



(b) AES/Rijndael, Hamming weight CPA (Pearson) last round.



(c) AES/Rijndael, Hamming distance CPA (Pearson) first round.



(d) AES/Rijndael, Hamming distance CPA (Pearson) last round.

Fig. 12. Second-order CPA attack on first and last round AES/Rijndael subkey (byte #1), where the correlation coefficient is shown on the vertical axis and the time samples during the relevant encryption rounds are shown on the horizontal axis.

S-Box input in the last round by predicting values $s_i = \text{S-box}^{-1}(ct_i \oplus keyhypo_i)$ for some index $1 \leq i \leq 16$, key hypothesis $keyhypo_i \in [0, 255]$ and ciphertext byte ct_i. These predictions are used directly when assuming the Hamming weight leakage. For the Hamming distance leakage model, these predictions are furthermore XORed with the corresponding input/output bytes.

4.4.1. Results

We mount the attacks on a set of 1.25 million power traces. The results of the DPA attack (using t-test distinguisher) are shown in Fig. 11. Figs. 11(a) and 11(c) show the case when targeting S-box output in the first round, while Figs. 11(b) and 11(d) show the case when targeting the S-box input in the last round. In all cases, the correct key byte value (in black) is not distinguishable from the other candidates (in gray), so the attack fails in recovering the key.

Running the CPA attack yields similar results, as shown in Figs. 12(a) and 12(c) when targeting the first round, and in Figs. 12(b) and 12(d) when targeting the last round. Note that for these experiments, we process only samples in the interval corresponding to the first (resp. last) round, rather than the whole encryption.

4.5. Further experiments

In this subsection, we present additional experiments regarding the proposed countermeasures and the cipher design, and we summarize our results.

4.5.1. Importance of mask m_2

The value stored in the round register is protected by both S-box Decomposition and Boolean Masking (using mask m_2, see Fig. 5). Considering the CFGLUT based architecture, this masking does not require any extra resources. However, to provide more insight about the necessity of masking inside the decomposed S-box, we have measured one million power traces without it (fully protected encryption, except that mask m_2 is set to zeros), and performed the test vector leakage assessment as described earlier. In the case of AES/Rijndael, both first-order and second-order t-tests turn out very similar to the previously presented results. However, in the case of Serpent without the m_2 mask, we have encountered worsening, where the first-order t-test values reach as high as 10 (second-order t-test, however, still yields results similar to those presented earlier).

4.5.2. Necessary randomness

The serial configuration I/O of CFGLUTs allows for various reconfiguration strategies (as mentioned in Section 3.3). Furthermore, a particular number of pairs get swapped when recomputing the S-box decompositions (as mentioned in Section 2.3.1). Selected strategy may significantly affect the amount of resources necessary. In this experiment, we compare implementations where:

- either all S-boxes are reconfigured in parallel based on same random data, or every S-box is reconfigured separately based on its own random data,
- the decomposed S-boxes get modified by swapping either one or eight pairs of elements in the bijection mapping.

All four combinations, for both ciphers, perform equally in test vector leakage assessment based on 300,000 measured power traces.

5. Conclusion

In this paper, we describe and evaluate side-channel attack protected AES and Serpent implementations, which are based on an approach demonstrated by Sasdrich et al. [8] for the PRESENT cipher. These implementations utilize dynamic logic reconfiguration, which can be easily deployed in both FPGA and ASIC designs. We describe a method by means of which a generic substitution–permutation network can be protected against side-channel attacks, and we tailor the approach to a Xilinx Spartan-6 FPGA for the protection of both AES and Serpent.

We demonstrate the effectiveness of the implemented countermeasures by evaluating the side-channel leakage using Welch's t-test, with different combinations of countermeasures in place. We did not detect any significant first-order leakage from the protected versions of both AES and Serpent encryption implementations using one million power traces. Using the same power traces, we detected apparent second-order leakage from Serpent encryption, while AES encryption second-order leakage is barely detectable. Furthermore, to provide more confidence about the implementation resilience, we attempted at breaking the protected AES implementation using second-order DPA and CPA attacks targeting both first and last round. All these attacks fail with 1.25 million power traces available.

Declaration of competing interest

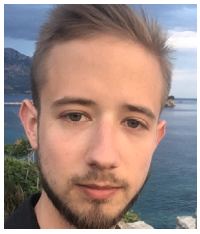
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

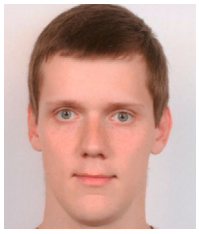
This work was partially funded by the Central Europe Leuven Strategic Alliance (CELSA) project "DRASTIC: Dynamically Reconfigurable Architectures for Side-channel analysis protection of Cryptographic implementations" (CELSA/17/033). Some authors were partially supported by the Czech Technical University (CTU) grants No. SGS17/213/OHK3/3T/18 and SGS20/211/OHK3/3T/18. Computational resources were partially supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

References

[1] P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in: M. Wiener (Ed.), Advances in Cryptology — CRYPTO' 99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 388–397.
[2] E. Brier, C. Clavier, F. Olivier, Correlation power analysis with a leakage model, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2004, pp. 16–29.
[3] J.-J. Quisquater, D. Samyde, Electromagnetic analysis (ema): Measures and counter-measures for smart cards, in: Smart Card Programming and Security, Springer, 2001, pp. 200–210.
[4] G. Piret, F.-X. Standaert, Security analysis of higher-order Boolean masking schemes for block ciphers (with conditions of perfect masking), IET Inf. Secur. 2 (1) (2008) 1–11.
[5] S. Nikova, C. Rechberger, V. Rijmen, Threshold implementations against side-channel attacks and glitches, in: International Conference on Information and Communications Security, Springer, 2006, pp. 529–545.
[6] D. Sokolov, J. Murphy, A. Bystrov, A. Yakovlev, Design and analysis of dual-rail circuits for security applications, IEEE Trans. Comput. 54 (4) (2005) 449–460.
[7] N. Mentens, B. Gierlichs, I. Verbauwhede, Power and fault analysis resistance in hardware through dynamic reconfiguration, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2008, pp. 346–362.
[8] P. Sasdrich, A. Moradi, O. Mischke, T. Güneysu, Achieving side-channel protection with dynamic logic reconfiguration on modern FPGAs, in: 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), IEEE, 2015, pp. 130–136.
[9] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J. Robshaw, Y. Seurin, C. Vikkelsoe, PRESENT: An ultra-lightweight block cipher, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2007, pp. 450–466.
[10] J. Daemen, V. Rijmen, AES Proposal: Rijndael, 1999.
[11] E. Biham, R. Anderson, L. Knudsen, Serpent: A new block cipher proposal, in: International Workshop on Fast Software Encryption, Springer, 1998, pp. 222–238.
[12] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, E. Roback, Report on the development of the advanced encryption standard (AES), J. Res. Natl. Inst. Stand. Technol. 106 (3) (2001) 511.
[13] M. Alle, K. Varadarajan, A. Fell, N. Joseph, S. Das, P. Biswas, J. Chetia, A. Rao, S. Nandy, R. Narayan, Redefine: Runtime reconfigurable polymorphic asic, ACM Trans. Embedded Comput. Syst. (TECS) 9 (2) (2009) 1–48.
[14] S. Das, S. Nandy, A flexible crypto-system based upon the REDEFINE polymorphic asic architecture, Def. Sci. J. 62 (1) (2012) 25–31.
[15] Xilinx, Spartan-6 Libraries Guide for HDL Designs, [Online] Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/spartan6_hdl.pdf.
[16] F.M. Brown, Boolean Reasoning: The Logic of Boolean Equations, Springer Science & Business Media, 2012.
[17] H. Guntur, J. Ishii, A. Satoh, Side-channel attack user reference architecture board SAKURA-G, in: Consumer Electronics (GCCE), 2014 IEEE 3rd Global Conference on, IEEE, 2014, pp. 271–274.
[18] T. Schneider, A. Moradi, Leakage assessment methodology, J. Cryptogr. Eng. 6 (2) (2016) 85–99.
[19] F.-X. Standaert, How (not) to use welch's t-test in side-channel security evaluations, in: International Conference on Smart Card Research and Advanced Applications, Springer, 2018, pp. 65–79.
[20] T.S. Messerges, Using second-order power analysis to attack DPA resistant software, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2000, pp. 238–251.
[21] S. Chari, C.S. Jutla, J.R. Rao, P. Rohatgi, Towards sound approaches to counteract power-analysis attacks, in: Annual International Cryptology Conference, Springer, 1999, pp. 398–412.
[22] E. Prouff, M. Rivain, Masking against side-channel attacks: A formal security proof, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2013, pp. 142–159.



Petr Socha received his master in embedded system design from Czech Technical University in Prague in 2019. Currently, Petr is a Ph.D. student at the Faculty of Information Technology, Czech Technical University in Prague. His research interests include digital design, cryptographic hardware, and side-channel security.



Jan Brejník is a student at Czech Technical University in Prague, Faculty of Information Technology. His research interests are digital design and side-channel security. He also works as an embedded software developer.

Josep Balasch obtained a joint Ph.D. degree from KU Leuven and Radboud University Nijmegen in 2014. Between 2014 and 2019 he was postdoctoral researcher at the COSIC research group, KU Leuven. He is currently visiting professor at the eMedia Lab, Faculty of Engineering Technology, KU Leuven. His research interests are in the area of embedded security, particularly on design methods and hardware/software architectures for cryptographic implementations. Josep has co-authored more than 35 publications in peer-reviewed journals and conferences, and serves in the program committee of several venues on embedded security.

Martin Novotný graduated in electrical engineering from the Czech Technical University in Prague, the Czech Republic, in 1992. He received his Ph.D. degree in information security from Ruhr-University Bochum, Germany, in 2009. Currently, he is an assistant professor and the head of the Embedded Security Lab at the Czech Technical University in Prague. His research interests include arithmetic units, hardware for cryptography and cryptanalysis, efficient implementations of cryptographic algorithms, and embedded systems. Martin serves as a program committee member in several international conferences focusing on cryptography and digital design. He was a program co-chair of DSD 2017, program chair of DSD 2018, and a general chair of CARDIS 2019 conferences. He is an author or co-author of 60+ journal and conference papers and book chapters.

Nele Mentens received her master and Ph.D. degree from KU Leuven in 2003 and 2007, respectively. Currently, Nele is a professor at Leiden University and KU Leuven. Her research interests are in the domains of hardware security and configurable computing. Nele was a visiting researcher for three months at Ruhr University Bochum in 2013 and at EPFL in 2017. She was/is the PI in around 15 finished and ongoing research projects with national and international funding. She serves as a program committee member of renowned international conferences on security and hardware design, such as NDSS, CHES, DAC, DATE, ESSCIRC and FPL. She was the general co-chair of FPL in 2017 and the program chair of FPL and CARDIS in 2020. Nele is (co-)author in over 100 publications in international journals, conferences and books. She won best paper awards and nominations at DATE'16, AsianHOST'17 and CHES'19. Nele is an associate editor of IEEE Transactions on Information Forensics and Security and IEEE Circuits and Systems Magazine. She is a Senior Member of the IEEE and serves as an expert for the European Commission.