Wire-Tap Codes as Side-Channel Countermeasure
– an FPGA-based experiment –

Amir Moradi

Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
amir.moradi@rub.de

Abstract. In order to provide security against side-channel attacks a masking scheme which makes use of *wire-tap codes* has recently been proposed. The scheme benefits from the features of binary linear codes, and its application to AES has been presented in the seminal article. In this work – with respect to the underlying scheme – we re-iterate the fundamental operations of the AES cipher in a hopefully more understandable terminology. Considering an FPGA platform we address the challenges each AES operation incurs in terms of implementation complexity. We show different scenarios on how to realize the SubBytes operation as the most critical issue is to deal with the large S-boxes encoded by the underlying scheme. Constructing various designs to actualize a full AES-128 encryption engine of the scheme, we provide practical side-channel evaluations based on traces collected from a Spartan-6 FPGA platform. As a result, we show that – despite nice features of the scheme – with respect to its area and power overhead its advantages are very marginal unless its fault-detection ability is also being employed.

1 Introduction

Nowadays security of embedded devices does rely not only on the underlying modern cipher but also on the way it is implemented. Side-channel analysis (SCA) attacks, which have been brought to the attention of scientific communities since the late 90s [14, 15], are amongst the major threats against the security of cryptographic devices. Amongst other countermeasures masking [6, 7, 17], which by randomizing the secret internals aims at cutting the relation between the side-channel leakages and predictable processes, is the most studied one. It can also be seen as the one that achieving its goals in practice is most challenging. For instance, dealing with glitches in hardware platforms is of major concerns for most of the masking schemes (see [18, 19, 21, 23]). Along the same line different masking approaches like additive [6], multiplicative [1], and affine [9, 26] have been introduced. It has been shown in [9] that affine masking, which combines both additive and multiplicative masking, can achieve – in terms of the number of traces – a considerably high level of security.

Recently a masking scheme which makes use of wire-tap channel concept has been proposed [5]. The scheme which is developed based on the principle of binary linear codes can be seen as an extended version of affine masking. More

precisely, in affine masking the bit size of the words, e.g., 8 in case of the AES state bytes, does not change after masking a secret. But in wire-tap approach the length of the masked value is expanded. Another specification of the scheme is regarding the selection of the masks. In affine masking the additive mask has the same length as the unmasked value, and also can take all possible values, e.g., $\{0,1\}^8$ in case of 8-bit AES state bytes. However, they are restricted to the elements of the underlying code (so-called *codewords*) in case of wire-tap code approach. An expanded value is seen as error bits which are added to a codeword, i.e., the mask. This therefore provides two features for the scheme:

- The additive mask can be removed without knowing it; thanks to parity-check matrix of the underlying code which can eliminate the codewords.
- Certain faults which cause the additive mask to be not a codeword anymore can be detected while the main goal of the scheme is to provide security against e.g., power analysis attacks.

The focus of [5] is on an AES encryption engine, and a procedure on how to make a protected version of the cipher under the proposed scheme is introduced. Moreover, by means of simulation results the success rate of first- and second-order CPA [4] attacks as well as MIA [10] for different settings is examined.

Our contribution in this work is in the direction of realizing a hardware implementation of the scheme for a certain setting. We first shortly restate the scheme and its features by concentrating on the AES-128 encryption as the target cipher. With respect to how to implement each operation of the cipher, we try to provide more details that are partially missing in the original work [5]. We also give solution for a couple of issues which we faced during the implementation that hugely affect its performance as well as its security. Based on practical investigations, which are performed on a Spartan-6 FPGA as the implementation platform, we provide a comparison between the underlying scheme and classical Boolean masking from efficiency and security points of view. In short, we show that the additional features that the scheme provides are very marginal considering its high performance overhead when it is implemented correctly.

2 Underlying Scheme

Since the theory behind the wire-tap channel [22, 27] and the secrecy of the scheme is given in detail in [5], below we mainly focus on the target cipher, i.e., AES-128 encryption, with respect to side-channel protection. For the sake of consistency we also try to follow the notations given in [5].

2.1 Notations

Suppose a binary linear code C of length n, dimension k, and minimum distance d as $[n,k,d]$. C consists in a generator matrix G of size $(n-k)\times n$ with rows of $(g_1,\ldots,g_{n-k})$, where each row is an n-bit binary vector. A codeword $c \in C$ is generated as $m \cdot G$, where m is an $(n-k)$-bit binary vector. Indeed all

2^{n-k} codewords of C can be generated by m going through the entire space vector $\{0,1\}^{n-k}$. C also contains a $k \times n$ (binary) parity-check matrix H as $\forall c \in C,\ c \cdot H^T = 0$.

In order to encode a k-bit message x it should be first expanded by means of a $k \times n$ matrix L as $x \cdot L$. Rows of $L = (l_1, \ldots, l_k)$ are n-bit binary vectors which are linearly independent of each other, and none of them is a codeword. The encoding of x is determined by adding (modulo 2) a randomly selected codeword to the result of the expansion as

$$z \ = \ x \cdot L \ \oplus \ m \cdot G.$$

At anytime the encoded message z can be multiplied by H^T to eliminate the added random codeword:

$$z \cdot H^T \ = \ x \cdot L \cdot H^T \oplus m \cdot \underbrace{G \cdot H^T}_{:=0}.$$

The result, i.e., $x \cdot L \cdot H^T$, which is in some cases called syndrome, is related to x. Indeed the decoding (deriving x from z) can be done if $x \to x \cdot L \cdot H^T$ forms a bijection.

2.2 Settings

Considering AES each message x is mapped to a cipher state byte as $k = 8$. Hereafter for the sake of simplicity without losing generality we specify the length of the code $n = 16$ as the evaluations reported in [5] indicate its high level of security. In other words, each state byte of AES is expanded to 16 bits and a $[16, 8, 5]$ binary code [11] is used for masking. This leads to generator matrix $G = \Big(- M^T | I_8 \Big)$ and parity-check matrix $H = \Big(I_8 | M \Big)$, where I_8 denotes the identity matrix of size 8×8 and

$$M = \begin{pmatrix} 10001011 \\ 11000101 \\ 11100010 \\ 01110001 \\ 10111000 \\ 01011100 \\ 00101110 \\ 00010111 \end{pmatrix}.$$

For other values of n and parametric definitions of G and H sizes see [5].

By defining the underlying code as above the only missing point is how to define the L matrix. An example of L given in [5] as $L_{\text{simple}} = (l_1, \ldots, l_k)$ is

chosen as

$$\begin{aligned}
l_1 &= [1000000000000000],\\
l_2 &= [0100000000000000],\\
l_3 &= [0010000000000000],\\
l_4 &= [0001000000000000],\\
l_5 &= [0000100000000000],\\
l_6 &= [0000010000000000],\\
l_7 &= [0000001000000000],\\
l_8 &= [0000000100000000].
\end{aligned}$$

Expanding x by L_{simple} results in x as the left half and 0 as the right half. Indeed, L_{simple} brings a feature as $x \cdot L \cdot H^T = x$. In other words, decoding of $z = x \cdot L \oplus m \cdot G$ is easily done by $z \cdot H^T := x$.

Further, a simple algorithm is given in [5] to randomly generate the L matrix. This resulted in an example (given in [5]) as $L_{\text{specific}} = (l_1, \ldots, l_k)$ as follows:

$$\begin{aligned}
l_1 &= [0111110110100101],\\
l_2 &= [1011000010000101],\\
l_3 &= [0001011100100111],\\
l_4 &= [0001110011001101],\\
l_5 &= [0000110101001100],\\
l_6 &= [1011110110111111],\\
l_7 &= [1001111111111111],\\
l_8 &= [0101100101110101].
\end{aligned}$$

However, decoding $z = x \cdot L_{\text{specific}}$ by H^T is not possible. It means that $x \to x \cdot L_{\text{specific}} \cdot H^T$ is not a bijection as

$$\exists x_1, x_2;\; x_1 \cdot L_{\text{specific}} \cdot H^T = x_2 \cdot L_{\text{specific}} \cdot H^T.$$

It in fact prevents L_{specific} to be considered as a valid case of L. In short, examining whether $x \to x \cdot L \cdot H^T$ is bijective is missing in the algorithm given in [5]. Alternatively it can be checked whether $L \cdot H^T$ has a right inverse Inv as

$$L \cdot H^T \cdot Inv \;=\; I_8.$$

Since Inv is an 8×8 binary matrix, its existence can be checked by an exhaustive search for each column separately, in sum in a space of 8×2^8.

As a valid example, replacing l_4 by [1010011111010111] causes L_{specific} to fulfill all the requirements. Hereafter we consider this corrected matrix as $L_{\text{specific}'}$ for further investigations. Also, an example for the corresponding Inv is given in Appendix.

2.3 AES Operations

XOR (AddRoundKey) Suppose x_1 and x_2 are encoded as $z_{i\in\{0,1\}} = x_i \cdot L_{\text{specific}'} \oplus c_i$ by two randomly selected codewords c_1 and c_2. A correct encoding of $x_1 \oplus x_2$ can be made by $z_1 \oplus z_2 = (x_1 \oplus x_2) \cdot L_{\text{specific}'} \oplus (c_1 \oplus c_2)$ since as a property of binary linear codes $(c_1 \oplus c_2)$ is also a codeword.

S-box (SubBytes) If $z = x \cdot L_{\text{specific}'} \oplus c$ is the input of the S-box, $z' = S(x) \cdot L_{\text{specific}'} \oplus c'$ as an encoding of $S(x)$ can be the result of a table lookup $S'(z)$, where $\forall x$ the look-up table S' is precomputed given two independent codewords c and c'.

Permutation (ShiftRows) Since ShiftRows is a byte-wise permutation, the corresponding word-wise permutation of an encoded cipher state results in correct encoding of the cipher state after ShiftRows.

Multiply by 2 MixColumns over a column (x_1, x_2, x_3, x_4) can be realized by an XOR sequence of the input bytes, some multiplied by 2 and 3 in GF(2^8). Since $x * 3 = (x * 2) \oplus x$, and as explained above XOR of the encoded words is straightforward, the only remaining operation to realize MixColumns is

$$\text{MUL2}(z) \;=\; (x * 2) \cdot L_{\text{specific}'} \oplus c''; \;\; z = x \cdot L_{\text{specific}'} \oplus c; \;\; c, c'' \in C.$$

A solution that we provide here (not clearly given in [5]) is to find a matrix P such that

$$x \cdot L_{\text{specific}'} \cdot P = (x * 2) \cdot L_{\text{specific}'}.$$

A simple solution is to select $P = L^{-1}_{\text{specific}'} \cdot M2 \cdot L_{\text{specific}'}$, where $L^{-1}_{\text{specific}'}$ denotes the right inverse of $L_{\text{specific}'}$ which as explained before can be found in a space of 8×2^{16}. Also $M2$ stands for binary matrix representation of multiply-by-2 in GF(2^8). An example for $L^{-1}_{\text{specific}'}$, $M2$, and P are given in Appendix.

Multiplying z by P leads to

$$z \cdot P \;=\; x \cdot L_{\text{specific}'} \cdot P \oplus c \cdot P \;=\; (x * 2) \cdot L_{\text{specific}'} \oplus c \cdot P.$$

Here the problem is that $c \cdot P$ is not necessarily a codeword. As a solution, also somehow followed by [5], $c'' \oplus c \cdot P$ should be added to the above result to obtain[1]

$$z \cdot P \oplus c'' \oplus c \cdot P \;=\; (x * 2) \cdot L_{\text{specific}'} \oplus c'' \;=\; \text{MUL2}(z).$$

A question arising here is whether there exists an L such that

$$\overset{?}{\exists}\, L, \; \forall c \in C; \;\; c \cdot P \in C.$$

The answer is unfortunately negative as if $c \cdot P$ is a codeword, applying the parity-check matrix should lead to

$$c \cdot P \cdot H^T \;=\; c \cdot L^{-1} \cdot M2 \cdot L \cdot H^T \;=\; 0.$$

However, $L \cdot H^T$ cannot be 0 following the definition of L (see Section 2.1). So, there is no way to avoid mask correction, i.e., adding $c'' \oplus c \cdot P$.

Algorithm 1 gives an overview of the full AES-128 encryption based on the selected settings and according to the one given in [5]. As explained above and

[1] Note that c'' and c must not be necessarily different.

Algorithm 1: AES-128 encryption protected by wire-tap code

Input : Plaintext $\mathcal{X}$, seen as 16 bytes x_i, $i \in \{0,\dots,15\}$,
11 RoundKeys $\mathcal{R}^j$, each seen as 16 bytes r_i^j, $j \in \{0,\dots,10\}$

Output: Ciphertext $\mathcal{Y}$, seen as 16 bytes y_i

1. Select 64 random bytes $m_{i\in\{0,\dots,63\}}$
2. Generate 4 vectors $\mathcal{C}$, $\mathcal{C}'$, $\mathcal{C}^1$, $\mathcal{C}^3$, each seen as 16 codewords as $c_i = m_i \cdot G$, $c_i' = m_{i+16} \cdot G$, $c_i^1 = m_{i+32} \cdot G$, $c_i^3 = m_{i+48} \cdot G$, $i \in \{0,\dots,15\}$
3. $\mathcal{C}^2 = \mathcal{C} \oplus \mathcal{C}^1$
4. $\mathcal{T} = \mathcal{C}^1 \oplus \texttt{MixColumn}\big(\texttt{ShiftRows}(\mathcal{C}')\big)$ `/* Mask Correction */`
5. **for** $i \in [\![0,15]\!]$ **do**
6. $\quad\forall x \in \{0,1\}^8;\ S_i'(x \cdot L_{\text{specific}'} \oplus c_i) = S(x) \cdot L_{\text{specific}'} \oplus c_i'$
7. $\quad z_i = x_i \cdot L_{\text{specific}'} \oplus c_i^1$ `/* Plaintext Encoding as` $\mathcal{Z}$ `*/`
8. $\quad\forall j \in [\![0,10]\!];\ k_i^j = r_i^j \cdot L_{\text{specific}'} \oplus c_i^2$ `/* RoundKey Encoding */`
9. **end**
10. **for** $j \in [\![0,9]\!]$ **do**
11. $\quad\forall i \in [\![0,15]\!]; z_i = z_i \oplus k_i^j$ `/*` $\mathcal{Z} : \big(\mathcal{X} \oplus \mathcal{R}^j\big) \cdot L_{\text{specific}'} \oplus \mathcal{C}$ `*/`
12. $\quad\forall i \in [\![0,15]\!]; z_i = S_i'(z_i)$ `/*` $\mathcal{Z} : \texttt{SB}\,\big(\mathcal{X} \oplus \mathcal{R}^j\big) \cdot L_{\text{specific}'} \oplus \mathcal{C}'$ `*/`
13. $\quad\mathcal{Z}=\texttt{ShiftRows}(\mathcal{Z})$ `/*` $\mathcal{Z} : \texttt{SR}\big(\texttt{SB}\,\big(\mathcal{X} \oplus \mathcal{R}^j\big)\big) \cdot L_{\text{specific}'} \oplus \texttt{SR}(\mathcal{C}')$ `*/`
14. $\quad$**if** $j \neq 9$ **then**
15. $\quad\quad\mathcal{Z}=\texttt{MixColumns}(\mathcal{Z})$ `/*` $\mathcal{Z} : \texttt{MC}\Big(\texttt{SR}\big(\texttt{SB}(\mathcal{X} \oplus \mathcal{R}^j)\big)\Big) \cdot L_{\text{specific}'} \oplus \texttt{MC}\big(\texttt{SR}(\mathcal{C}')\big)$ `*/`
16. $\quad\quad\mathcal{Z} = \mathcal{Z} \oplus \mathcal{T}$ `/*` $\mathcal{Z} : \texttt{MC}\Big(\texttt{SR}\big(\texttt{SB}\,(\mathcal{X} \oplus \mathcal{R}^j)\big)\Big) \cdot L_{\text{specific}'} \oplus \mathcal{C}^1$ `*/`
17. $\quad$**else**
18. $\quad\quad\mathcal{Z} = \mathcal{Z} \oplus \mathcal{C}^3$ `/*` $\mathcal{Z} : \texttt{SR}\big(\texttt{SB}\,(\mathcal{X} \oplus \mathcal{R}^9)\big) \cdot L_{\text{specific}'} \oplus \texttt{SR}(\mathcal{C}') \oplus \mathcal{C}^3$ `*/`
19. $\quad$**end**
20. **end**
21. $\forall i \in [\![0,15]\!]; z_i = z_i \oplus k_i^{10}$ `/*` $\mathcal{Z} : \mathcal{Y} \cdot L_{\text{specific}'} \oplus \texttt{SR}(\mathcal{C}') \oplus \mathcal{C}^3 \oplus \mathcal{C}^2$ `*/`
22. $\forall i \in [\![0,15]\!]; w_i = z_i \cdot H^T$ `/*` $\mathcal{W} : \mathcal{Y} \cdot L_{\text{specific}'} \cdot H^T$ `*/`
23. $\forall i \in [\![0,15]\!]; y_i = w_i \cdot Inv$ `/*` $\mathcal{Y}$ `: ciphertext */`

also as given in line 16 of the algorithm, at every round the masks should be corrected to keep them as valid codewords since MixColumns is not transparent to the underlying code. This in fact limits one of the main features of the scheme as the same mask correction is usually done for a classical Boolean masking scheme (e.g., see DPA contest V4 [25]). Note that the required mask correction is not due to our selection of $L_{\text{specific}'}$ or matrix P. The same scenario is given by the algorithm available in the original work of [5].

The only place in the algorithm which makes use of this feature – as masks can be removed without knowing them – is the last step of the algorithm lines 22 and 23. Indeed these two lines can be replaced by

$$\mathcal{Y} = (\mathcal{Z} \oplus \texttt{SR}(\mathcal{C}') \oplus \mathcal{C}^3 \oplus \mathcal{C}^2) \cdot L^{-1}_{\text{specific}'},$$

where first the mask is removed then the right inverse of $L_{\text{specific}'}$ is applied to obtain the ciphertext. This way the masks do not need to be valid codewords.

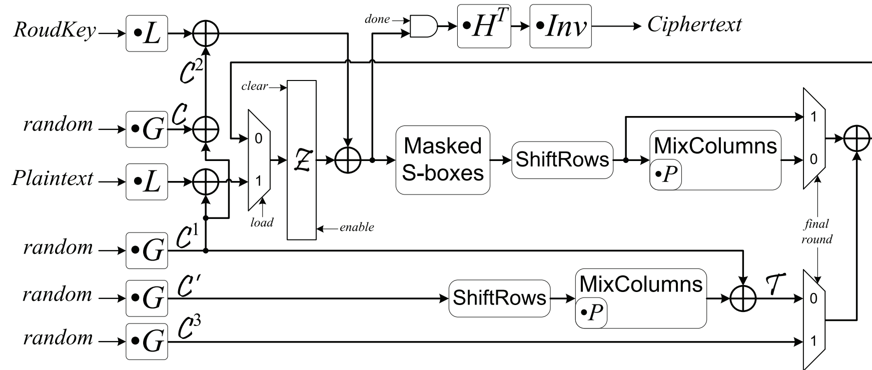

Fig. 1. Overall architecture of the design

So, any $c \in \{0,1\}^{16}$ can be selected as a mask to form $\mathcal{C}$, $\mathcal{C}'$, $\mathcal{C}^1$, and $\mathcal{C}^3$. More precisely, no binary linear code is required to be chosen, and any L – with a right inverse – can be employed to extend the cipher state bytes. It indeed shows that this feature of the scheme has a very marginal application for this perspective.

3 Hardware Design

Regardless of the limiting issue mentioned above, we give the details of the hardware design we developed to realize Algorithm 1. The target platform chosen for our practical experiments is a Spartan-6 LX75 FPGA embedded on a Side-channel AttacK User Reference Architecture (SAKURA-G) [20]. Due to the nature of the algorithm and efficiency as the main feature of hardware platforms, we aim at a round-based architecture, i.e., all operations of a cipher round are performed in parallel. This usually leads to a design with "single clock per round" capability. However, as it is explained later the efficiency that can be reached by the underlying scheme might be much less.

Figure 1 shows an overview of the hardware design which is well matched with Algorithm 1. It should be noted that the RoundKeys $\mathcal{R}$ have been precomputed. As shown in the algorithm and also by the design diagram, the mask of the round input, i.e., $\mathcal{C}^1$, is constant during an encryption. It is essential since for the selected masks the masked S-boxes S' are precomputed once at the start of the encryption. This might be a dangerous situation for a hardware platform as bit flips of the registers – modeled by Hamming distance (HD) – affect the amount of power consumption. Indeed if two values masked by the same mask, $x \oplus m$ and $y \oplus m$, are consecutively saved in a register, $x \oplus m \oplus y \oplus m = x \oplus y$ has an influence on power consumption which might be easily detected by $\text{HD}(x, y)$. Therefore, there are two important facts which should be considered for the design:

– The most problematic case of the above mentioned issue is at the last cipher round. Due to the absence of MixColumns, the XOR result of two consecutive

values stored by the state register $\mathcal{Z}$ is $\mathtt{SB}^{-1}\Big(\mathtt{SR}^{-1}\left(\mathcal{Y} \oplus \mathcal{R}^{10}\right)\Big) \oplus \mathcal{Y}$ expanded by $L_{\text{specific}'}$. Therefore, by means of a different mask at the end of the last round (see lines 16 and 18 of Algorithm 1) the mask of the ciphertext, i.e., $\mathtt{SR}\left(\mathcal{C}'\right) \oplus \mathcal{C}^3 \oplus \mathcal{C}^2$, is different to the mask of the last round input, i.e., $\mathcal{C}^1$. This prevents the addressed problem[2].

- Still the mask of the input of the first two rounds are the same. It may give an opportunity to a side-channel adversary to control the plaintexts thereby simplifying the prediction of consecutive values stored by certain words of the state register $\mathcal{Z}$. Therefore, in order to prevent this the state register should be precharged before saving a new entry at each round (see *clear* signal of state register $\mathcal{Z}$ in Fig. 1).

3.1 Masked S-box

The most challenging part of the design is how to actualize the masked S-boxes S'. The problem is due to the bit length of the S-box input, i.e., 16 bits in our settings, which causes realizing a table with 16×2^{16} bits problematic. In case of our platform there exist 172 instances of 18-kbit Block RAMs (BRAM)[3], 64 of them are required to make this table. However, it consists of only 256 valid entries for the chosen input and output masks.

Direct Mapping A straightforward solution is to map the encoded S-box input to an 8-bit (masked) value and directly perform the table lookup over a 256-entry table containing the encoded S-box outputs (size of $16 \times 2^8 = 4$ kbits fitting into a 9-kbit BRAM). In order to do so, one can add $m \cdot L_{\text{specific}'}$ to the encoded S-box input as

$$(x \cdot L_{\text{specific}'} \oplus c) \oplus m \cdot L_{\text{specific}'},$$

where $c = m \cdot G$. Now by applying the H^T matrix and Inv as defined in Section 2.2 we obtain

$$((x \oplus m) \cdot L_{\text{specific}'} \oplus m \cdot G) \cdot H^T \cdot Inv \;=\; \left((x \oplus m) \cdot L_{\text{specific}'} \cdot H^T\right) \cdot Inv \;=\; x \oplus m.$$

Therefore, we can convert the encoded S-box input to its corresponding Boolean masked value. However, this is in contradiction with the concept of the underlying scheme since the operation is performed on classical Boolean masked data not under the defined binary linear code.

Binary Search Another solution given in [5] is to make two tables, one to save 16-bit encoded S-box inputs and the other for the corresponding 16-bit encoded S-box outputs. So, a space of $32 \times 2^8 = 8$ kbits is required for these two tables

[2] Indeed, introducing a new mask as $\mathcal{C}^3$ is not required, but to keep the algorithm compatible with that of [5] we kept it in the design.

[3] Each 18-kbit BRAM can be configured as two independent 9-kbit BRAM resulting in total 344 instances. For more information see `RAMB16BWER` and `RAMB8BWER` in [28].

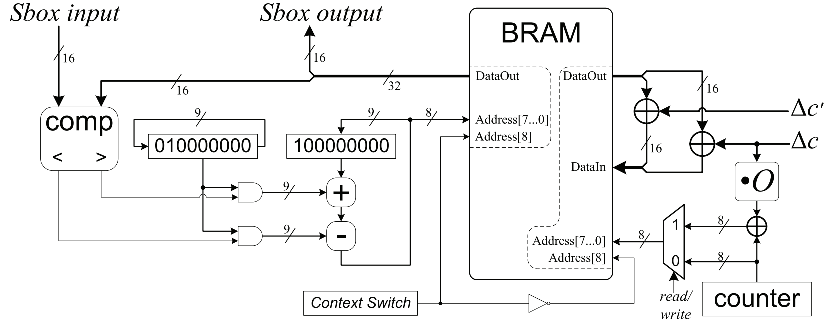

Fig. 2. Block diagram of the masked S-box circuit (left) binary search module, (right) update module

which can easily fit into a half of a 18-kbit BRAM. Therefore, for each masked S-box lookup the input table should be searched.

Since the search through an arbitrary table might be very inefficient, we considered the binary search which requires the tables to be sorted based on the encoded input. Figure 2 shows a block diagram of the binary search module. The BRAM is configured to use a 32-bit data path for input and output and 8 bits for the address[4]. Each 32-bit entry of the BRAM contains a tuple of (input, output) as $(x \cdot L_{\text{specific}'} \oplus c,\ S(x) \cdot L_{\text{specific}'} \oplus c')$. At the start of the search, `10000000` is given to the address of the BRAM. During the next clock cycle comparison result of the given encoded S-box input and the 16-bit part of the BRAM output decides whether `01000000` should be added or subtracted from the previous BRAM address. This procedure is continued till the comparison shows equality. In the worst case it takes 9 clock cycles to obtain the desired encoded S-box output[5].

In contrast to **Direct Mapping**, in this case the address of the BRAM does not necessarily form a classical Boolean masking of the input x as $x \oplus m$. But since it is the index of the table sorted based on the encoded S-box input, it can be modeled as $f(x) \oplus g(m)$, where

- $f(x)$ stands for a function which maps x to the index of the sorted table of $x \cdot L_{\text{specific}'}$, but
- deriving $g(m)$ is more complicated and explained in Appendix.

Both these two functions are deterministic based on $L_{\text{specific}'}$. They are also linear and can be represented by a matrix multiplication as $g(m) = m \cdot G \cdot O$, where O is the sorting matrix given in Appendix. Compared to **Direct Mapping**, a second-order SCA attack with a HD/HW model might be less efficient supposing that $f(\cdot)$ and $g(\cdot)$ are not known. However, it does not generally increase the

[4] Indeed the address has 9 bits width, we explain later how to control the 9th bit.

[5] It can be done also in 8 clock cycles, but the BRAM content of address `10000000` should be previously saved in a separate register.

robustness of the implementation against a second-order side-channel adversary as long as they do not randomly change. It can be seen as with the same outcome as **Direct Mapping** that the encoded S-box input is converted to a Boolean masked version for the table lookup.

The use of this approach brings one more consequence. According to [28] each BRAM has a register for the address input and an optional register for the data output. A conceptual block diagram of a Xilinx BRAM is given in [3]. The problem here is that during the binary search the BRAM address register as well as its output register (if enabled regardless of the selected option) stores the consecutive values which are masked by the same mask. As explained before, this leads to a side-channel leakage depending on the XOR result of the corresponding two unmasked values (for a similar observation see [2]). In case of the address register $f(x_1) \oplus f(x_2) = f(x_1 \oplus x_2)$ has a considerable contribution on the amount of power consumption when $f(x_1) \oplus g(m)$ and $f(x_2) \oplus g(m)$ are consecutive addresses given to the BRAM by the binary search module. The same holds for the output register as the difference between two consecutive (input, output) tuples is $\Big((x_1 \oplus x_2) \cdot L_{\text{specific}'},\ \big(S(x_1) \oplus S(x_2)\big) \cdot L_{\text{specific}'} \Big)$. Note that even if the output register is not enabled, the output signals still drive a combinatorial circuit (here MixColumns) and the same leakage can be seen.

In order to avoid such leakage, similar to the state register $\mathcal{Z}$ the BRAM also should be precharged during the binary search process. In our designs we have precharged the BRAM address by 0 before giving it a new address. Therefore, it leads to maximum 18 clock cycles for each masked S-box lookup. Due to the existence of a register for the BRAM input, precharge of the state register $\mathcal{Z}$ can be performed at the same time with the last clock cycle of the binary search. It leads to maximum 19 clock cycles per cipher round, i.e., $190 + 1$ for one complete encryption[6]. This is extremely higher than that of the **Direct Mapping** approach, i.e., $20 + 1$ clock cycles for a complete encryption. At each cipher round, one clock cycle is needed to save in the state register $\mathcal{Z}$ and precharge the BRAM, and one clock cycle for both the mask S-box lookup and precharge of the state register.

In short both above expressed approaches have the same drawback as the S-box lookup needs to be out of the underlying code. Note that it can be prevented only if the (input, output) table is randomly – independent of m – permuted, which causes a search to take in average 128 clock cycles (at most 256).

3.2 Mask Update

As stated before, all the masks – derived from the 64 random bytes m_i – stay unchanged during the whole encryption. Moreover, the same holds for $\mathcal{C}^2$ derived from other masks as well as for $\mathcal{T}$ used for mask corrections at the end of each round (see Algorithm 1). By giving a new set of m_i new codewords and constants should be computed. Also, the masked S-box tables S' need to be updated. A

[6] One last additional clock cycle is required to save the final round output in the state register $\mathcal{Z}$ (see Fig. 1).

scheme to use BRAMs as the masked tables and dynamically update them is previously given in [13] called *BRAM Scrambling*. It is mainly based on the dual-port feature of BRAMs and the fact that two tables can fit in one BRAM. Fortunately it is the case for our settings as for both approaches given above two masked tables simply fit into a BRAM. Two 4-kbit tables fit into each 9-kbit BRAM in case of **Direct Mapping**, and one 18-kbit BRAM can hold two 8-kbit tables of **Binary Search** approach.

While the encryption module uses the first half of the BRAM via one of the ports, the other port is used to make the second half updated. After finishing the update process when an encryption is not in progress the context switch is performed thereby using the updated second half for the encryption and updating the first half. Note that the ports used for encryption and update are not swapped. Generally only one bit as the MSB of the 9-bit address changes to switch the halves used by these two modules.

Based on the two approaches given in Section 3.1 the corresponding update modules are slightly different. In case of **Direct Mapping**, it is similar to the one explained in [13]. Suppose m as the random byte from which input mask $c = m \cdot G$ has been derived, and suppose c' as the output mask. The new randoms are denoted by m_{new}, respectively c_{new} and c'_{new}. The XOR of the current and new masks are given to the update module as $\Delta m = m \oplus m_{\text{new}}$ and $\Delta c' = c' \oplus c'_{\text{new}}$.

The update module performs the following operations

- It reads the address $i := x \oplus m$ from one half of the BRAM (the part being used for encryption) as $v := S(x) \cdot L_{\text{specific}'} \oplus c'$.
- It saves $v \oplus \Delta c' := S(x) \cdot L_{\text{specific}'} \oplus c'_{\text{new}}$ at the address $i \oplus \Delta m := x \oplus m_{\text{new}}$ of the other half of the BRAM.

The above procedure is repeated 256 times $\forall i \in [\![0, 255]\!]$ to finish the update process.

For the **Binary Search** approach the update module is a bit more complicated since the updated table should be saved as sorted. Here $\Delta c = c \oplus c_{\text{new}}$ is also given to the update module, and the following operations are performed:

- The address $i := f(x) \oplus g(m)$ is read from one half of the BRAM as $(w,\ v) := \big(x \cdot L_{\text{specific}'} \oplus c,\ S\left(x\right) \cdot L_{\text{specific}'} \oplus c'\big)$.
- $(w \oplus \Delta c,\ v \oplus \Delta c') := (x \cdot L_{\text{specific}'} \oplus c_{\text{new}},\ S\left(x\right) \cdot L_{\text{specific}'} \oplus c'_{\text{new}})$ is stored at the address $i \oplus g(\Delta m) := f(x) \oplus g(m_{\text{new}})$ of another half of the BRAM.

$f(\cdot)$ and $g(\cdot)$ are those functions introduced in Section 3.1. As explained before, $g(m) = m \cdot G \cdot O$, where details of sorting matrix O are given in Appendix. Here $g(\Delta m)$ can be computed as

$$\Delta m \cdot G \cdot O \;=\; (m \oplus m_{\text{new}}) \cdot G \cdot O \;=\; \Delta c \cdot O.$$

A block diagram of the update module for the **Binary Search** approach is shown by Fig. 2. In both cases the update process can be done in 512 clock cycles since each read or write operation needs to be done in a separate clock

cycle. Therefore, similar to the scheme presented in [13] – depending on the delay between consecutively given plaintexts – the masks used for a couple of encryptions may not be different.

4 Security Evaluation

4.1 Evaluation

Before providing practical evaluation results, we would like to comment on the security evaluations presented in [5]. If an arbitrary byte x_1 (and x_2) is encoded by all possible 256 codewords we obtain a set $S_1 = \{x_1 \cdot L_{\text{specific}'} \oplus m \cdot G | m \in \{0,1\}^8\}$ (and respectively S_2). As a feature of the scheme – thanks to the parity-check matrix – the masks can be removed without knowing them. However, it means that S_1 and S_2 do not have any overlap, i.e., $\nexists z;\ z \in S_1,\ z \in S_2$. Therefore, if the SCA leakages associated to S_1 and S_2 are different, their observation may lead to categorize them based on x_1 and x_2. This means that under certain assumptions an SCA attack will be possible. This assumptions are related to the order of the attack as well as the algebraic degree of the pseudo-Boolean representation of the leakage function. It is indeed the same concept as *low-entropy masking* studied in [16] and [12]. Since the selected code in our settings is an optimal $[16, 8, 5]$ binary code, it can resist against first-order attacks if the algebraic degree of the leakage function $d < 5$. Formally speaking

$$\mathbb{E}\big(\mathscr{L}\,(X \cdot L_{\text{specific}'} \oplus M \cdot G) \mid X = x\big)$$

is constant for any pseudo-Boolean function $\mathscr{L}$ of algebraic degree $d < 5$, where X represents any 8-bit random variable, M is a random variable uniformly distributed on $\{0,1\}^8$, and $\mathbb{E}$ stands for expectation. Similarly it can resist against univariate second-order attacks (without considering the leakage of the mask m or $m \cdot G$) for all leakage functions with algebraic degree $d < 3$. However, a univariate third-order attack works if the leakage function is not linear.

In theory a univariate mutual information analysis (MIA) [10] should be able to distinguish x considering the distribution of $\mathscr{L}\,(x \cdot L_{\text{specific}'} \oplus M \cdot G)$. The higher the algebraic degree of $\mathscr{L}$ is, the easier MIA can distinguish x. In fact, these issues have not been considered in the evaluations of [5], and a noisy Hamming weight (HW) model, i.e., a linear $\mathscr{L}$, is considered to show the resistance of the scheme against first-order attacks using simulation results.

4.2 Implementation

For practical experiments we developed three implementations:

- *Profile 1*, with the design of Fig. 1 and $L_{\text{specific}'}$ where the masked S-boxes are realized by **Binary Search** approach,
- *Profile 2*, the same as *Profile 1* except the masked S-boxes which are actualized by **Direct Mapping** technique,

Table 1. Comparison of area and performance of three designed profiles

Profile	**Area**			**Performance** (# clock)			
	# Register	# LUT	# BRAM 9k*	Round		Total	
				avg**	max	avg	max
1	3 050	14 499	32	17	19	171	191
2	2 747	13 808	16	2	2	21	21
3	1 595	2 320	16	2	2	21	21

* In case of *Profile 1*, each 18-kbit BRAM is counted as two 9-kbit BRAMs.
** Average performance of a binary search over an N-element array is $O(\log_2(N) - 1)$.

– *Profile 3*, with the same design as the other profiles, but without using the wire-tap approach. In other words, it follows the architecture of Fig. 1, but realizes a classical Boolean masking, i.e., no expansion and no binary linear code is used. The masked S-boxes are also implemented straightforwardly similar to that of *Profile 2* but without any mapping.

As stated before, our implementation platform is a Spartan-6 LX75 FPGA of SAKURA-G. Table 1 represents a comparison of area and performance of our three profiles. It should be emphasized that the area required for KeySchedule and to store the RoundKeys is ignored in the given numbers. Also, the PRNG module which is supposed to provide random numbers is excluded in the comparisons as all three profiles need the same PRNG and the same number of random bytes per encryption. As the last note, *Profiles 1* and *2* are designed to be parametric, i.e., for any selected L and C with length $n = 16$ the area requirements stay unchanged.

As expected, compared to the others *Profile 1* is the largest and slowest design. Also, comparing *Profiles 2* and *3* the overhead of using the wire-tap approach based on our settings becomes clear, i.e., 0.7 times more registers and 4.95 times more logic LUTs.

4.3 Measurements

In order to perform practical evaluations we collected the power traces of the target FPGA by means of a LeCroy digital oscilloscope at the sampling rate of 1 GS/s. The measurements are done by monitoring the voltage drop by a 0.6 Ω resistor placed at the Vdd path. We also made use of the amplifier embedded on SAKURA-G to increase the level of the signal compared to the electrical noise level. During the measurements the target FPGA is clocked at a frequency of 3 MHz.

As illustrated in Section 3.1, from a security point of view *Profiles 1* and *2* are roughly the same, and evaluation of *Profile 1* faces more challenges due it is much longer traces. Therefore, we provide here only the evaluation result of *Profiles 2* and *3*, a sample trace of each is shown by Fig. 3. As expected, due to its higher area requirements *Profile 2* has much higher power consumption

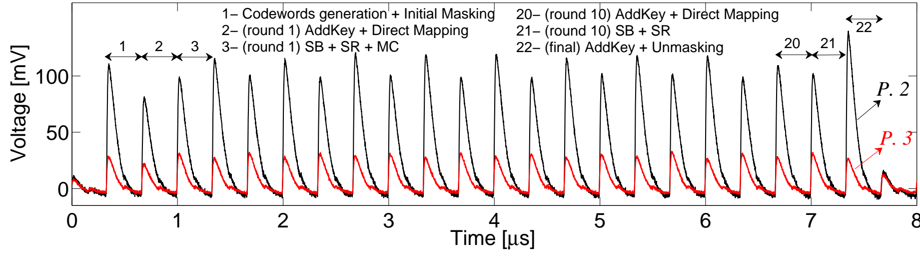

Fig. 3. Sample traces of *Profile 2* and *Profile 3*

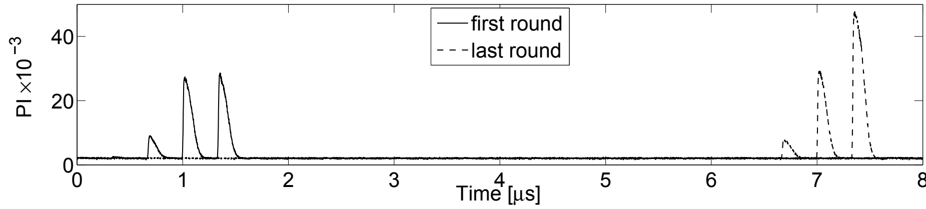

Fig. 4. *Profile 2*, PRNG off, Perceived Information curves based on an S-box output byte of first and last rounds using 1 000 000 traces

compared to *Profile 3*. As the evaluation result of *Profile 1* is given in Appendix, a sample trace of *Profile 1* is shown by Fig. 7.

PRNG off In order to have a reference about the leakage of our platform as well as to verify our measurement setup we first considered *Profile 2* when the PRNG is switched off. Therefore, all 64 bytes $m_{i\in\{0,\ldots,63\}}$ (see Algorithm 1) are constant as 0 during the collection of 1 000 000 traces. In other words, in this case only the expansion with $L_{\text{specific}'}$ is performed and all codewords used for encoding are selected as 0.

Our security evaluations are based on the *Information Theoretic* (IT) metric of [24]. It means that we estimate the mutual information between the measured traces and a secret internal which we suppose to know. In order to prevent missing any statistical moments, in all our evaluations the probability distributions are estimated by histograms of 12 bins rather than Gaussian. Following the notations of [24], we estimate the mutual information as

$$\mathrm{I}(S;\mathbf{L}) = \mathrm{H}[S] + \sum_{s} \Pr[s] \sum_{\mathbf{l}} \Pr[\mathbf{l}|s] \cdot \log_2 \Pr[s|\mathbf{l}],$$

where S (a secret internal) is selected as an S-box output in our evaluations[7]. We indeed measure the amount of perceived information [8] as we estimate the

[7] Due to the bijective property of AES S-box and the former linear key addition, it indeed leads to the same result if the corresponding plaintext byte is selected.

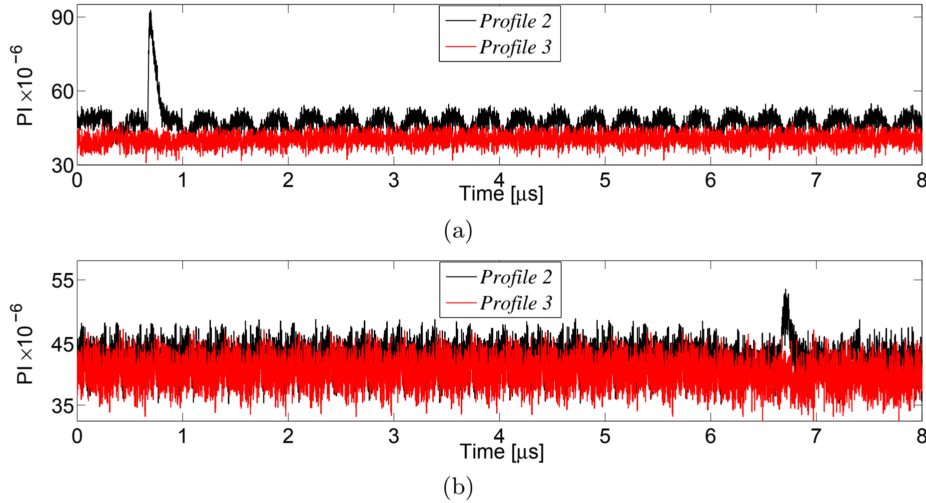

Fig. 5. *Profiles 2* and *3*, PRNG on, Perceived Information curves based on an S-box output byte of (a) first and (b) last round using 50 000 000 traces

distributions by a histogram and consider the leakage model based on an S-box output value.

Figure 4 shows two perceived information curves each of which associated to an S-box output; one for the first round and the other for the last round of encryption. As expected – due to constant masks – information available through the traces either at the first round or at the last round can be easily detected.

PRNG on By switching the PRNG on we first confirmed the uniform distribution of 64 random bytes given to the design. As stated before, due to a long time required to update the masked S-boxes some encryptions may share the same masks. Therefore, we kept a considerable delay between consecutive encryptions during the measurements in order to make sure that no mask is reused. In other words, one encryption does not start till the last mask update process is finished. In this settings we collected 50 000 000 traces from each of *Profiles 2* and *3*.

Repeating the last experiment as estimating the perceived information between the measured traces and the S-box output led to the curves shown in Fig. 5(a) and Fig. 5(b) for the first and the last round respectively. As shown by the graphics, there is a difference between the estimated perceived information of *Profile 2* and *Profile 3*. It indeed confirms our statement in Section 4.1 about the existence of a univariate leakage in case of *Profile 2* though due to its very low magnitude a practical attack might be very challenging. Note that the same result is observed by evaluation of *Profile 1* as shown in Fig. 7 in Appendix.

Attacking an implementation realized by BRAMs is in general harder compared to the corresponding circuit purely implemented by combinatorial elements (see [3]). It becomes more challenging if the implementation is equipped

with masking as in the case of our profiles. We should stress that in all of our profiles the BRAMs' ports used for encryption are disabled right after finishing the encryption process; it can be seen by the sample traces of Fig. 3. Therefore, the leakage observed in [2] are not seen here. Moreover, since we precharge the address register of BRAMs by 0, their output register do not store a deterministic value as the S-box table is masked. This also hardens a key-recover attack.

In order to perform a higher-order analysis the leakages of the mask and masked data should be combined. It is actually referred as *second-order analysis* in [5] where the centered-product of simulated leakages (noisy HW) of c and $x \cdot L \oplus c$ is taken as the combined leakage. As expressed before in Section 3.2, by giving a new set of random bytes at the start of an encryption the masks and all other constants are computed. The related leakage is observable only at the first clock cycle (see Fig. 3). Since these values stay unchanged during an encryption, their corresponding leakage do not combine with the leakage associated to the process of masked data. Therefore, we should manually do the combination required for higher-order analyses. So, we performed the following steps:

– We first made all traces mean free. The means at each sample point is computed based on the selected target value, i.e., an S-box output of the first or the last round.
– Next, we obtained the average of a part of a mean-free trace related to the first clock cycle (see Fig. 3).
– At the last step the average value is multiplied to all points of the mean-free trace.

The last two steps are repeated for all mean-free traces independently, and the second step is needed since the leakage associated to the masks may not appear at a specific sample point.

Finally, we repeated the same experiment as before by estimating the perceived information between the preprocessed traces and an S-box output at both the first and the last round. The corresponding results shown by Fig. 6 indicate the same observation as before, i.e., there exist more exploitable leakages by *Profile 2* compared to *Profile 3*. We should highlight that although the magnitude of perceived information compared to that of Fig. 5 is increased, performing a successful key-recovery attack on both of these two profiles is not an easy task. This is due to the high level of switching noise related to our round-based architectures as well as to the random precharge of BRAM output registers.

5 Conclusions

In this work we have taken an in-depth look at the wire-tap coding approach as a side-channel countermeasure with focus on AES and an FPGA as the target platform. Under certain assumptions and settings we have demonstrated the difficulties a hardware designer may face when implementing the basic modules of the cipher. The most challenging issue is how to realize the masked S-box, that

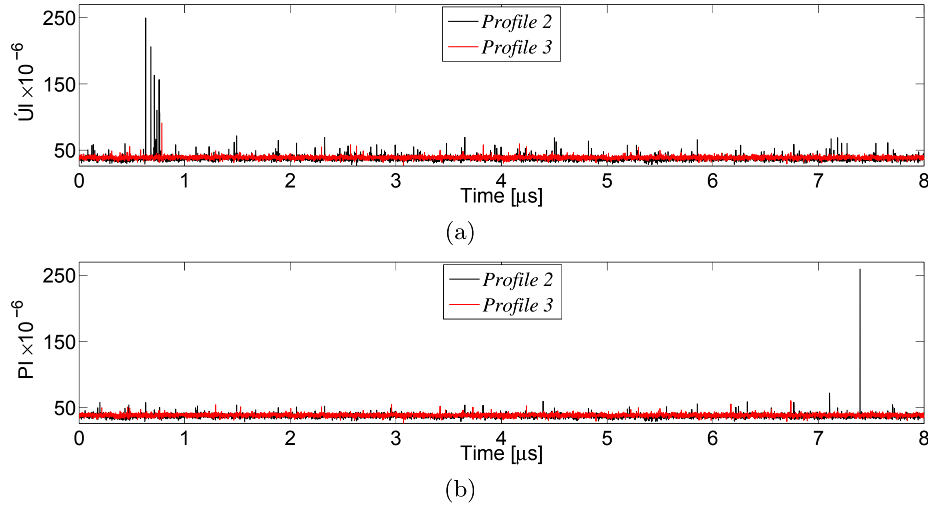

Fig. 6. *Profiles 2* and *3*, PRNG on, Perceived Information curves based on an S-box output byte of (a) first and (b) last round using 50 000 000 center-product preprocessed traces

is due to the S-box size and the underlying binary code length. As the encoded S-box cannot easily fit into the memory, we examined a couple of solutions, most of which turns the design into a sort of classical Boolean masking for S-box lookup. The problems we addressed here can be mitigated for other ciphers with a smaller S-box size, where the whole encoded S-box can be straightforwardly implemented as a look-up table.

We have shown that one of the nice features of the scheme as *the possibility to unmask without the knowledge of the mask* is only beneficial at the end of the cipher operations, which can be replaced by a simple XOR. Moreover, due to the intransparency of MixColumns to the underlying binary code the mask correction is unavoidable during the cipher-round computations. These two issues cause the scheme to be not much advantageous compared to classical Boolean masking. Further, our practical implementations on a Spartan-6 FPGA showed a considerable area overhead, i.e., 0.7 times more registers and 5 times more LUTs, compared to a corresponding design of classical Boolean masking which expectedly consumed less energy. Our practical side-channel analyses also indicated that the underlying scheme does not provide a higher level of resistance. Indeed, we have shown that the scheme might be vulnerable to certain attacks while the corresponding Boolean masking design is still robust.

In short, with respect to only power analysis and compared to Boolean masking we do not find a motivating advantage of the scheme – in our settings – as the circuit is more complicated, needs more energy, and is slightly less robust against power analysis attacks. However, we have not considered two advantage of the scheme in our analyses:

– Due to the properties of the binary linear codes, a higher-order version of the scheme can be made with moderate efforts.
– Expansion matrix L can randomly change. This results in requiring another module responsible for generation of a new L following its requirements. Other matrices like P to multiply by 2, Inv to decode the ciphertext, sorting matrix O, and etc should also be dynamically obtained accordingly. If so, another source of randomness is needed for the system which optimistically complicates a key-recovery attacks.
– The scheme by nature can detect certain faults. Protecting against both DPA and fault attacks is of crucial interest as most of the known countermeasures can deal with only one of them. By the extensive evaluation given here we provided first a roadmap how to implement it, and second an overview about its DPA resistance in practice. As a result, this scheme might be a potential candidate to increase security against both DPA and fault attacks, that the later one should be carefully investigated to determine to which extend it can defeat fault attacks.

Acknowledgment

The author would like to thank Julien Bringer from Morpho (France), Stefan Heyse, Cornel Reuber, and Tobias Schneider from Ruhr University Bochum (Germany) for their helpful discussions and comments.

References

1. M.-L. Akkar and C. Giraud. An Implementation of DES and AES, Secure against Some Attacks. In *CHES 2001*, volume 2162 of *LNCS*, pages 309–318. Springer, 2001.
2. S. Bhasin, S. Guilley, A. Heuser, and J.-L. Danger. From cryptography to hardware: analyzing and protecting embedded Xilinx BRAM for cryptographic applications. *J. Cryptographic Engineering*, 3(4):213–225, 2013.
3. S. Bhasin, W. He, S. Guilley, and J.-L. Danger. Exploiting FPGA block memories for protected cryptographic implementations. In *ReCoSoC 2013*, pages 1–8. IEEE, 2013.
4. E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In *CHES 2004*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.
5. J. Bringer, H. Chabanne, and T. H. Le. Protecting AES against side-channel analysis using wire-tap codes. *J. Cryptographic Engineering*, 2(2):129–141, 2012.
6. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
7. J.-S. Coron and L. Goubin. On Boolean and Arithmetic Masking against Differential Power Analysis. In *CHES 2000*, volume 1965 of *LNCS*, pages 231–237. Springer, 2000.
8. F. Durvaux, F. Standaert, and N. Veyrat-Charvillon. How to Certify the Leakage of a Chip? In *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 459–476. Springer, 2014.

9. G. Fumaroli, A. Martinelli, E. Prouff, and M. Rivain. Affine Masking against Higher-Order Side Channel Analysis. In *SAC 2010*, volume 6544 of *LNCS*, pages 262–280. Springer, 2010.
10. B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel. Mutual Information Analysis. In *CHES 2008*, volume 5154 of *LNCS*, pages 426–442. Springer, 2008.
11. M. Grassl. Code Tables: Bounds on the Parameters of Various Types of Codes. `http://www.codetables.de/`, June 2008.
12. V. Grosso, F. Standaert, and E. Prouff. Low Entropy Masking Schemes, Revisited. In *CARDIS 2013*, volume 8419 of *LNCS*, pages 33–43. Springer, 2013.
13. T. Güneysu and A. Moradi. Generic Side-Channel Countermeasures for Reconfigurable Devices. In *CHES 2011*, volume 6917 of *LNCS*, pages 33–48. Springer, 2011.
14. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO 1996*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
15. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
16. H. Maghrebi, S. Guilley, and J.-L. Danger. Leakage Squeezing Countermeasure against High-Order Attacks. In *WISTP 2011*, volume 6633 of *LNCS*, pages 208–223. Springer, 2011.
17. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
18. S. Mangard, N. Pramstaller, and E. Oswald. Successfully Attacking Masked AES Hardware Implementations. In *CHES 2005*, volume 3659 of *LNCS*, pages 157–171. Springer, 2005.
19. A. Moradi, O. Mischke, and T. Eisenbarth. Correlation-Enhanced Power Analysis Collision Attack. In *CHES 2010*, volume 6225 of *LNCS*, pages 125–139. Springer, 2010.
20. Morita Tech. Side-channel AttacK User Reference Architecture (SAKURA). Further information are available via `http://www.morita-tech.co.jp/SAKURA/en/index.html`.
21. S. Nikova, V. Rijmen, and M. Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *J. Cryptology*, 24(2):292–321, 2011.
22. L. H. Ozarow and A. D. Wyner. Wire-Tap Channel II. In *EUROCRYPT 1984*, volume 209 of *LNCS*, pages 33–50. Springer, 1985.
23. E. Prouff and T. Roche. Higher-Order Glitches Free Implementation of the AES Using Secure Multi-party Computation Protocols. In *CHES 2011*, volume 6917 of *LNCS*, pages 63–78. Springer, 2011.
24. F.-X. Standaert, T. Malkin, and M. Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 443–461. Springer, 2009.
25. TELECOM ParisTech SEN research group. DPA Contest (4th edition), 2013-2014. `http://www.DPAcontest.org/v4/`.
26. M. von Willich. A Technique with an Information-Theoretic Basis for Protecting Secret Data from Differential Power Attacks. In *Cryptography and Coding, IMA Int. Conf.*, volume 2260 of *LNCS*, pages 44–62. Springer, 2001.
27. A. D. Wyner. The Wire-Tap Channel. *Bell System Technical Journal*, 54(8):1355–1387, 1975.
28. Xilinx. Spartan-6 Libraries Guide for HDL Designs. Available via `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/spartan6_hdl.pdf`, April 2012.

Appendix

Matrices

A right inverse of $L_{\text{specific}'}$, and Inv as a right inverse of $(L_{\text{specific}'} \cdot H^T)$:

$$L^{-1}_{\text{specific}'} = \begin{pmatrix} 01111011 \\ 11110011 \\ 11011001 \\ 00010010 \\ 11100111 \\ 00011001 \\ 11101001 \\ 01110111 \\ 10011111 \\ 00110110 \\ 11000001 \\ 11101000 \\ 11111111 \\ 01001000 \\ 00011011 \\ 00100111 \end{pmatrix}, \qquad Inv = \begin{pmatrix} 01100110 \\ 10001100 \\ 01111101 \\ 00010011 \\ 00110011 \\ 01000111 \\ 11011110 \\ 11011101 \end{pmatrix}.$$

GF(2^8) multiply-by-2 binary matrix $M2$, P as the matrix for multiplication by 2 with respect to $L_{\text{specific}'}$ and underlying code C, and sorting matrix O (the corresponding algorithm is given below):

$$M2 = \begin{pmatrix} 00011011 \\ 10000000 \\ 01000000 \\ 00100000 \\ 00010000 \\ 00001000 \\ 00000100 \\ 00000010 \end{pmatrix}, \qquad P = \begin{pmatrix} 0101111110010000 \\ 1001010001010110 \\ 0011111010111011 \\ 1010101010011000 \\ 1000111000111101 \\ 0010111100001111 \\ 1001100100011001 \\ 1111010100001011 \\ 1111001111101101 \\ 0001011101010001 \\ 1000111001001011 \\ 0000011011100110 \\ 0011111011001101 \\ 1101101001110010 \\ 1001001010110000 \\ 1001111110001001 \end{pmatrix}, \qquad O = \begin{pmatrix} 10000000 \\ 01000000 \\ 00100000 \\ 00010000 \\ 00001000 \\ 00000100 \\ 00000000 \\ 00000000 \\ 00000000 \\ 00000010 \\ 00000000 \\ 00000000 \\ 00000000 \\ 00000000 \\ 00000001 \\ 00000000 \end{pmatrix}.$$

$M2$ is selected as $\forall x : (x_1, \ldots, x_8) \in \{0,1\}^8;\ (x_1, \ldots, x_8) \cdot M2 = x * 2$, where x_1 and x_8 are selected as MSB and LSB of x respectively.

Sorting Matrix

To find the sorting matrix O we need function $f(\cdot)$, which has been defined in Section 3.1. $\forall x \in \{0,1\}^8$, $f(x)$ gives the index of the sorted table $x \cdot L_{\text{specific}'}$. We indeed require its inverse as $f^{-1}(f(x)) = x$. The matrix O can be obtained by following Algorithm 2.

Algorithm 2: How to obtain sorting matrix O

Input : $L_{\text{specific}'}$ as the expansion matrix,
$f(\cdot)$ as indexes of a sorted 256-entry table of $x \cdot L_{\text{specific}'}$ and its inverse as $f^{-1}(\cdot)$

Output: Sorting matrix O with rows of $(o_1, \ldots, o_{16})$, where each row is an 8-bit binary vector,
as $f(x) \oplus c \cdot O$ gives the index of the sorted table $x \cdot L_{\text{specific}'} \oplus c$, where $c = m \cdot G$

1 **for** $i \in [\![1, 16]\!]$ **do**
2 $\quad$ $index_{\min} = \arg\min_{x} \left(x \cdot L_{\text{specific}'} \oplus 2^{i-1}\right)$
3 $\quad$ o_i=binary representation of $f^{-1}(index_{\min})$
4 **end**

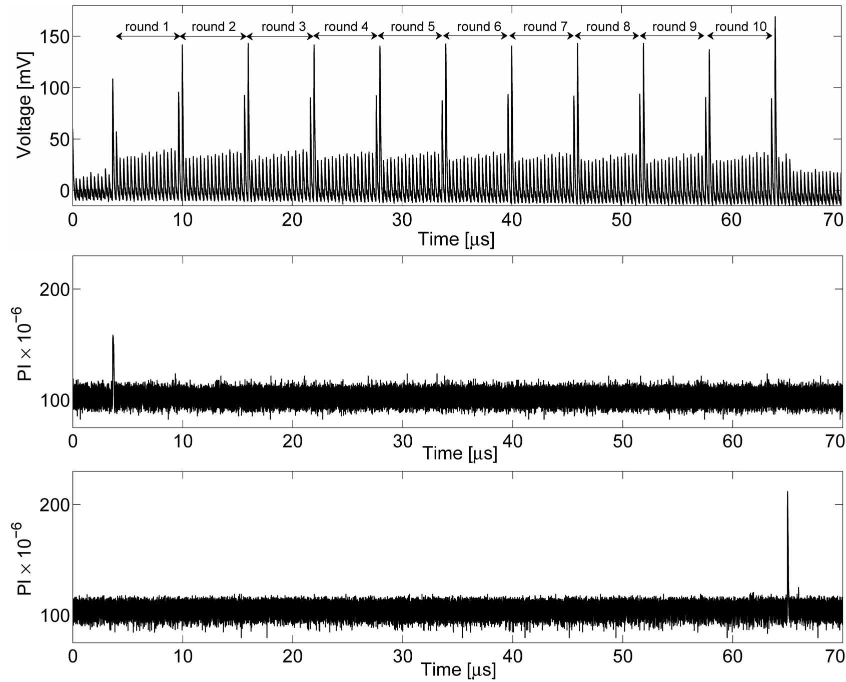

Fig. 7. *Profile 1*, PRNG on, (top) a sample trace, and Perceived Info. curves based on an S-box output byte of (middle) first and (bottom) last rounds using 20 000 000 traces