Algebraic Side-Channel Attacks on Masked Implementations of AES

Luk Bettale, Emmanuelle Dottax and Mailody Ramphort*
Idemia, France

Keywords: Algebraic Side-Channel Attack, Masking, AES, Template Attack, SAT Solving.

Abstract: Algebraic Side-Channel Attacks allow an attacker to exploit single trace leakages in an automated way. The literature mentions the fact that these attacks have the potential to defeat the masking countermeasure. Though, this context has not been explored a lot and the lack of experiments makes it difficult to evaluate the feasibility of these attacks in practice. We set-up a framework to perform such attacks and made new experiments on state-of-the-art masking schemes. We focused on the number of leakages required for an attack, and considered realistic leakage points. Our experiments and analyses allow to precisely estimate the minimal number of leakages required for a successful key recovery.

1 INTRODUCTION

Algebraic Side-Channel Attacks (ASCA) have been introduced by (Renauld and Standaert, 2009). Their idea was to combine the techniques used in algebraic cryptanalysis (Courtois and Pieprzyk, 2002) with side-channel analysis, with the intent of exploiting the leakages from a limited number of executions. Indeed, few leakages may bring too little information for classical side-channel analysis. ASCA proposes to use an improved, more elaborated offline cryptanalysis step, in an attempt to fully exploit the available information. The principle is to write the target cipher as a system of equations, and to add equations related to the leakages. For instance, equations stating the Hamming weight of some intermediate results, as recovered by the side-channel analysis step. This system of equations is then solved during the offline phase. In most cases, the system is converted into a Boolean satisfiability (SAT) problem and a SAT-solver is used to *automatically* recover the key bytes, as described in e.g. (Bard et al., 2007), (Courtois and Bard, 2007). Other types of solvers have also been analysed. For instance, (Carlet et al., 2012) considered solvers based on Gröbner bases.

ASCA allow the exploitation of information extracted from as little as a single trace. Such extractions can be achieved via profiling side-channel attacks: the adversary is allowed to use an open copy on the final target to tune his attack. He then tries the attack on the target device to extract information on manipulated data. Template attacks (Chari et al., 2003) are a typical example. This field has recently been given a second wind by the use of machine learning techniques. For instance, (Heuser and Zohner, 2012) showed that the use of machine learning techniques improves the attacks in the case of highly noisy traces. Many recent papers are devoted to this field, and the improvements they bring to template attacks argue in favour of ASCA.

In addition to requiring fewer observations to succeed, ASCA presents several interesting aspects. First, they can potentially exploit leakages from any intermediate variable, in any round. Furthermore, they can succeed in an unknown plaintext/ciphertext adversarial model. Another particularly interesting feature of ASCA is their capacity to defeat masking. Few papers in the literature have studied this point: (Renauld and Standaert, 2009) and (Renauld et al., 2009) present some results on PRESENT and AES respectively. However, no deep analysis has been conducted. In particular, no extensive experiments have been made to determine the conditions when these attacks work, and the efficiency of ASCA in the masked context has thus not been thoroughly evaluated.

In this article, we aim at exploring deeply the capabilities of ASCA on masked implementations of AES. To this end, we have set-up a framework to perform ASCA on AES. We consider the best attacker model, where the leakages measured are always correct. The progresses currently made in the template analysis field speak in favour of considering this con-

*This work has been done while this author was with Oberthur Technologies (now Idemia).

Bettale, L., Dottax, E. and Ramphort, M.
Algebraic Side-Channel Attacks on Masked Implementations of AES.
DOI: 10.5220/0006869502580269
In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications (ICETE 2018) - Volume 2: SECRYPT*, pages 258-269
ISBN: 978-989-758-319-3

text. Thus, we did not use data from any actual device, but rather used simulation to generate leakages. We have validated our set-up by first reproducing attacks on unmasked implementations by (Renauld et al., 2009) and (Mangard, 2003). We then use our framework on various state-of-the-art, masked versions. We analyse how each masking scheme impacts ASCA, look for the best leakage points, and try to find the minimal number of leakages required to break the masked implementation with ASCA. Furthermore, we also consider the whole attack, *i.e.*including the profiling phase. To this end, we tried ASCA with a limited number of different leakage points, and with leakage points that would be easier to characterize.

The remaining of this document is organised as follows. Section 2 provides a survey of the works published in the field of ASCA, and gives a focus to ASCA on masked implementations. Section 3 explains the choices we have made for our study, the tools we have used and how we have worked. Section 4 gives the results we obtained without any masking, and Section 5 the results on masked implementation. Section 6 concludes this document and gives some perspectives.

2 CONTEXT

2.1 ASCA

ASCA have been introduced by (Renauld and Standaert, 2009). They use the principle and the techniques of algebraic attacks (Courtois and Pieprzyk, 2002), but in a side-channel analysis context. Using algebraic techniques in the offline cryptanalysis step allows one to take advantage of a wider range of leakages. In particular, these attacks can take advantage of leakages in intermediate rounds, and they can succeed in an unknown plaintext/ciphertext adversarial context. Additionally, they proved to require fewer observations than traditional side-channel attacks to succeed.

The first (offline) step of ASCA, is to describe the target algorithm as a set of equations with the key bits as variables. Given such a representation, recovering the key bits amounts to solving the system of equations. In order to limit the degree of the equations and to get a system exploitable in practice, techniques similar to the ones by (Courtois and Pieprzyk, 2002) are used: additional, internal variables are introduced. Furthermore, the non-linear substitution is also represented in a way that limits the degree of the equations, for instance with the process described in (Biryukov and De Cannière, 2003).

Then, in the online phase, the target algorithm is executed and side-channel leakages are gathered. The additional information (plaintext values, ciphertext values, Hamming weights of intermediate variables) is written as equations and added to the existing system.

The offline phase consists in trying to solve the resulting system. In order to do so, various techniques might be used and the system of equations might have to be modified in order to fit the chosen technique. For instance, using SAT solvers requires converting the system into Conjunctive Normal Form (CNF) formulas.

The first work (Renauld and Standaert, 2009) focused on the cryptosystem PRESENT and was extended to AES in (Renauld et al., 2009). Several improvements regarding the algebraic representation of the leakages where proposed in (Mohamed et al., 2012; Oren et al., 2012; Carlet et al., 2012). The possibility to handle erroneous leakages has also been explored, either by describing this situation as an optimization problem (Oren et al., 2010; Oren and Wool, 2012), or by considering the leakages as a set of values (Song et al., 2014; Mohamed et al., 2012; Zhao et al., 2012). A different error-tolerant approach has been introduced in (Veyrat-Charvillon et al., 2014), that uses a code in place of algebraic equations. This method has been compared to ASCA in the case of error-free leakages in (Grosso and Standaert, 2015). In (Banciu and Oswald, 2014; Banciu et al., 2015), the authors compare the performances of ASCA and other single-trace attacks in the presence of faulty leakages.

2.2 Masking

Side-Channel Analysis takes advantage of statistical dependencies that exist between a physical leakage (e.g., the power consumption, the electromagnetic emanations) produced during the execution of a cryptographic algorithm, and the intermediate values manipulated. In particular, the principle of Differential Side-Channel Analysis (DSCA) is the following: the attacker executes the cryptographic algorithm several times with different inputs and gets a set of, say, power consumption traces, each trace being associated to one value known by the attacker. At some points in the algorithm execution, sensitive variables are manipulated: variables that can be expressed as a function of a secret variable and the known variable. The principle of DSCA is to make hypotheses on the value of the secret and deduce hypotheses on sensitive values. Statistical tools are then used to compute the correlation between the predictions and the acquired power traces. They allow the attacker to (in)validate his hypotheses, and recover the value of the secret.

Masking is the state of the art countermeasure against DSCA. This section gives some details regarding the different implementation techniques that one might encounter.

2.2.1 Generalities

A masked implementation is an implementation where every sensitive variable v is manipulated in a masked form, $v \oplus m$, where m is a uniformly distributed random number. Doing so, the correlation with the secret value is lost, and DSCA fails. Note that *higher order*-DSCA can be used to defeat masking: the attacker gets traces of both the manipulations of $v \oplus m$ and m and combines them to recover the secret value. A countermeasure to a DSCA targeting d intermediate values is to split v into $d+1$ values, using d random masks. In this work, we limit ourselves to first-order masking, where one mask is used.

The challenge of masking is to succeed in computing the expected result, while maintaining masks on sensitive variables. Any linear function L is easily implemented on masked data by using an additive masking, as $L(v \oplus m) = L(v) \oplus L(m)$. Masking non-linear parts is more challenging, and as a consequence a variety of techniques has been proposed. They are summarized in Section 2.2.2.

When masking a whole en/de-cryption process, different strategies might be applied regarding the number of masks and their management. We list hereafter some aspects of these strategies.

Mask Propagation vs Mask Correction. In a mask propagation scheme, the algorithm is applied to the masked variable without any change; the value of the resulting mask is computed separately, at each step. This often amounts to (at least) double the execution time of the algorithm. In a mask correction scheme, the mask is *corrected* at some step. This allows to save some computation time using pre-computation related to the mask that can be used at each step.

Single-mask Protection vs Multi-mask Protection. A k-bit CPU will only manipulate k-bit values at once. This means that a single mask with k-bit entropy is enough to mask a whole state at a given time. The single-mask protection uses the same k-bit value to mask separately all k-bit chunks of a state. This pushes the mask correction one step further as the pre-computations can be performed only once. In the multi-mask protection mode a whole state is masked with a full entropy mask. Note that, it does not make sense to use the single-mask protection with mask propagation as the propagation occurs on the whole state in a cipher. The three other combinations are possible.

All Rounds Masking vs Partial Rounds Masking. After each new round, every bit in the intermediate state depends on more key bits, forcing a standard DSCA to make more hypotheses. As masking schemes are resource and time consuming, one can consider removing the masking in the inner rounds. This is assumed safe when the number of key hypothesis is above a certain value. This strategy has been followed for constrained devices in, e.g. (Tillich et al., 2007).

2.2.2 Masking Non-linear Operations

Now we examine the different methods to manage masked variables in non-linear operations.

Table Re-computation Methods. Here, the non-linear operation is described as a substitution box (S-box) and one or several masked tables are computed, based on the standard S-box S and the additive mask. For instance, methods in (Chari et al., 1999), (Messerges, 2001) or (Akkar and Giraud, 2001) involve the pre-computation in RAM of a new table T such that $T(x) = S(x \oplus m) \oplus r$, where m is the input mask and r is the output mask (possibly equal to m). Each time the S-box has to be applied on the masked input $x' = x \oplus m$, the table T gives the masked result: $T(x') = S(x) \oplus r$. A variant has been proposed by (Prouff and Rivain, 2008), where the table is computed on-the-fly each time it has to be accessed. This method can use different input and output masks at virtually no cost, so it can be used in single-mask and multi-mask modes. One can note that both presented methods actually implement a mask correction. Another table-based countermeasure has been proposed in (Goubin and Patarin, 1999). It involves the pre-computation of two tables associated to the function $(x,y) \mapsto (A(x,y), S(x \oplus y) \oplus A(x,y))$, where A is a randomly chosen secret transformation. Accessing both tables with $(x \oplus m, m)$ as an input gives the new mask $A(x \oplus m, m)$ and the new masked value $S(x) \oplus A(x \oplus m, m)$. The paper proposes several variants to reduce the memory consumption.

S-box Secure Computation. In this case, the masked value $S(x) \oplus r$ is computed from the pair $(x \oplus m, m)$ via an algorithm parametrized by the 3-tuple $(x \oplus m, m, r)$. The computation of S is split into masked elementary operations (bitwise operations, addition, multiplication,...), and possibly by accessing one or several look-up table(s). In (Oswald et al., 2004; Oswald et al., 2005; Oswald and Schramm, 2006), composite field arithmetic is used: each element of $GF(256)$ is represented as a polynomial over $GF(16)$, and the AES S-box is computed with masked operations in $GF(16)$ only. As the AES S-box can be decomposed as a linear operation and a power function (actually, the inversion), another approach is to use multiplicative masking. Masking schemes based on this approach have been proposed by (Akkar and Giraud, 2001; Golic and Tymen, 2003; Trichina et al., 2003), but showed to be imperfect protections against DSCA. This method has been improved in (Genelle et al., 2010).

ISW-based Computation. In (Ishai et al., 2003), the authors introduced a generic method to mask a multiplicative operation that can be generalized at any order. Combined with a bit-sliced representation of the SBox (Rebeiro et al., 2006; Goudarzi and Rivain, 2016), this method allows to securely compute the AES SBox. For first-order masking on 8-bit devices, this method is less efficient and hence we do not consider it.

2.3 Masking and ASCA

ASCA have the theoretical power to defeat masking, as they apply to a single-trace setting. Indeed, the masking can be represented in the system of equations by adding some variables and equations for the masks. However, only few papers have considered making experiments with masked implementations.

The target of (Renauld and Standaert, 2009) is the block cipher PRESENT (Bogdanov et al., 2007). Different attack scenarios are considered: leakages for consecutive intermediate values, leakages for random intermediate values, known/unknown plaintext or ciphertext, and masked implementation. The masking scheme considered in the paper is a variant of the "duplication method", as proposed in (Goubin and Patarin, 1999). Let S from $\{0,1\}^n$ to $\{0,1\}^m$ be the S-box of the target algorithm. The technique consists in pre-computing the table associated to the function $A : (x,y) \longmapsto S(x) \oplus S(x \oplus y)$. Then, the new intermediate value $x' = S(x)$ can safely be computed via the standard S-box, the new mask m' being given by A: $m' = A(x \oplus m, m) = S(x \oplus m) \oplus S(x)$. One can notice that this amounts to applying the cipher algorithm unmodified on a masked input, while maintaining the mask value at each step. As far as ASCA is concerned, this is equivalent to considering the standard algorithm with unknown plaintext and ciphertext. An alternative approach is to build a system of equations that includes the masking scheme and to solve it with known plaintext and ciphertext. (Renauld and Standaert, 2009) shows that both strategies succeed in a single trace scenario.

Masked implementations of AES are mentioned in (Renauld et al., 2009), two different masking schemes are considered. The first one is the one proposed by (Oswald and Schramm, 2006), where the AES S-box is computed using a table for the multiplicative inverse in $GF(16)$. A multi-mask strategy is used, so that each byte of the state is masked with a different mask. Assuming they can obtain the Hamming weights of all intermediate computations, the attack succeeds. Due to the large amount of data available, the success is even greater than with an unmasked implementation. The second one is the one proposed by (Herbst et al., 2006). This method uses a table recomputation to mask the AES S-box, with different values to mask inputs and outputs. The same masks are used for every invocation of the S-box. This implementation is also vulnerable to ASCA, but the attack turned out to be more time consuming.

To our knowledge, no other experiments have been done on masked implementations. In particular, no detailed figures have been given regarding the minimal number of leakages required by the attack, or the best leakage points.

3 IMPLEMENTATIONS

In our experiments to evaluate the efficiency of ASCA against masked implementations, we make the following assumptions:

- The target runs on an 8-bit micro-controller.
- The target leaks the Hamming weight (HW) of the manipulated (8-bit) variables.
- The measures always give the exact HW with no noise.

The third assumption may seem unrealistic in general. However, on 8-bit platforms, this kind of precision has been showed to be achievable with enhanced template analysis or machine learning.

As a reference and to evaluate our framework, we will consider the key schedule (KS) of the AES, and a non-masked AES implementation. We will then consider three different masking schemes for the AES. We focus on the most relevant first-order masking schemes for AES, most widely found on embedded 8-bit implementations. We consider thus different implementations, that we summarize hereafter and which are described in detail in the following sections.

`KS`: The standard implementation of the key schedule. The modelling of this implementation is detailed in Section 3.1.

`plain`: No masking. The modelling of this implementation is detailed in Section 3.2.

`1Mask`: Each byte of the state is masked with the same byte throughout the execution. The SubBytes part is implemented with a table recomputation using the same mask as output, the MixColumns uses an additional temporary 1-byte mask, to avoid unmasking sensitive bytes. The modelling of this implementation is detailed in Section 3.3.1.

`1MaskGF16`: Each byte of the state is masked with the same byte throughout the execution, but the computation of the inverse is made in $GF(16)$. This is similar to the previous masking scheme, but the SubBytes part is decomposed as a mapping to $GF(16)^2$ and the inverse is performed in $GF(16)$ with a table re-computation of the inverse function with masked output. The modelling of this implementation is detailed in Section 3.3.2.

`16MaskGF16`: 16 bytes masking scheme with inverse computation in $GF(16)$. As the recomputed table is smaller in $GF(16)$, it is possible to precompute the inverse table for all possible values. In this case, no need to add a temporary mask during the MixColumn part, and a mask propagation is used for this step.

In the algorithms presented in this section, the potential leaking steps are indicated by the symbol ○(Xor) or ●(table lookup), and the total number of leakages in bytes (b), or in nibbles (n), is summarized for each operation.

3.1 Key Schedule (KS)

Attacking the key schedule of AES when Hamming weights of intermediate results can be extracted from the power trace of the device has been considered in (Mangard, 2003). We aim here at comparing the result of our ASCA framework to the ad-hoc method of the referred paper. The algorithm to derive one round key from the preceding is detailed in Alg. 1.

Our modelling of the key scheduling process uses $128 \times 10 = 1280$ extra variables for all the subkeys. Each subkey bit is simply described by the algebraic combination of the previous subkey.

Algorithm 1: Operations and leakages for the Key Scheduling of AES.

```
$k_0 \leftarrow k_0 \oplus S[k_{13}] \oplus \mathrm{RC}_r$        ▷ ○
$k_1 \leftarrow k_1 \oplus S[k_{14}]$                             ▷ ○
$k_2 \leftarrow k_2 \oplus S[k_{15}]$                             ▷ ○
$k_3 \leftarrow k_3 \oplus S[k_{12}]$                             ▷ ○
for $i = 4$ to 15 do                                              ▷ leak ×12b
    $k_i \leftarrow k_i \oplus k_{i-4}$                           ▷ ○
return $k$
```

3.2 Unprotected AES (plain)

As said, for reference we consider modelling the plain, unprotected AES. The algorithm is detailed in Alg. 2. It takes as input a 16-byte plaintext $(\mathrm{msg}_i)_{(i=0..15)}$ and a 16-byte key $(\mathrm{key}_i)_{(i=0..15)}$. The AES is implemented straightforwardly with the standard operations: AddRoundKey (ARK), SubBytes (SB), ShiftRows (SR) and MixColumns (MC). The leaked values are:

- each byte at the output of ARK,
- each byte at the output of SB,
- each intermediate byte during the MC computation.

Algorithm 2: Operations and leakages for the `plain` version of AES.

```
for $i = 0$ to 15 do
    $s_i \leftarrow \mathrm{msg}_i$
    $k_i \leftarrow \mathrm{key}_i$
for $r = 1$ to 10 do                                     ▷ Rounds main loop
    for $i = 0$ to 15 do                                 ▷ ARK: leak ×16b
        $s_i \leftarrow s_i \oplus k_i$                  ▷ ○
    $k \leftarrow \mathrm{NextSubKey}(k, r)$
    for $i = 0$ to 15 do                                 ▷ SB/SR: leak ×16b
        $s_i \leftarrow S'[s_i]$                         ▷ ●
        $s'_i \leftarrow s_{\rho(i)}$
    if $r < 10$ then
        for $i = 0$ to 12 by 4 do                        ▷ MC: leak ×52b
            $t \leftarrow s'_i \oplus s'_{i+1} \oplus s'_{i+2} \oplus s'_{i+3}$    ▷ ○
            for $j = 0$ to 3 do
                $u \leftarrow s'_{i+j} \oplus s'_{i+(j+1 \bmod 4)}$            ▷ ○
                $u \leftarrow 2u$                                             ▷ ○
                $s_{i+j} \leftarrow t \oplus s'_{i+j} \oplus u$               ▷ ○
for $i = 0$ to 15 do
    $\mathrm{ciph}_i \leftarrow s'_i \oplus k_i$
return ciph
```

The algebraic modelling of this algorithm only adds extra variables for the state at the beginning of each round which amounts to $128 \times 10 = 1280$ extra variables. As for the key scheduling, we describe the algebraic relation of each state bit as an algebraic equation of the bits of the previous step. The degree of the equation does not exceed 8, the algebraic degree of the S-Box. This approach is quite different from previous works, where the S-Box is often modelled to minimize the overall degree of the equations at the expense of high degree relations between the output bits (Renauld et al., 2009). Our approach leaves the output bits of the S-Box independent from one another.

However, the equations representing the Hamming weight of the leakages also have to be included. The problem is that these equations also have maximum degree 8. Plugging directly the S-Box output (degree 8) into these equations would result in equations with too large an amount of monomials. We decided to add extra variables for the intermediate values of the MixColumns, that is $52 \times 8 \times 9 = 3744$ extra variables. To link these new variables to the other ones, we also added some extra equations that represent each extra bit in function of the previous state, and each bit of the next state in function of the extra bits.

This modelling technique will also be used for masked implementations.

3.3 Masking Schemes

In this section, we review the three considered masking schemes. We present the modelling and the considered leaking points.

3.3.1 One Byte Masking Scheme (`1Mask`)

This masking scheme uses the single-mask protection in a mask correction setting. The algorithm implementing this masking scheme is detailed in Alg. 3. It takes as input a 16-byte plaintext $(\text{msg}_i)_{(i=0..15)}$ and a 16-byte key $(\text{key}_i)_{(i=0..15)}$. An initialization step consists in picking at random two masks. The first one, m_1, is the main mask. It is used to mask the input state, and to mask the inputs and outputs of the S-Box. A second mask m_2 is needed: it is used only temporarily in MixColumns, to prevent unmasking. The value $m_3 = m_1 \oplus m_2$ is computed and stored, so that it can be used at the end of MixColumns to recover values masked with m_1.

The modelling for this implementation is no different from the previous one. We only had to add 16 extra variables to represent the masks m_1 and m_2. These extra bits will appear in the equations describing the leakages.

Algorithm 3: Operations and leakages for the `1Mask` version of AES.

```
$m_1 \leftarrow$ rand                                   ▷ mask 1
$m_2 \leftarrow$ rand                                   ▷ mask 2
$m_3 \leftarrow m_2 \oplus 2m_2$                        ▷ mask correction MC
$S' \leftarrow x \mapsto S[x \oplus m_1] \oplus m_1$    ▷ masked S-Box gen.

for $i = 0$ to 15 do
    $s_i \leftarrow \text{msg}_i \oplus m_1$             ▷ mask msg
    $k_i \leftarrow \text{key}_i$
for $r = 1$ to 10 do                                    ▷ Rounds main loop
    for $i = 0$ to 15 do                                ▷ ARK: leak ×16b
        $s_i \leftarrow s_i \oplus k_i$                  ▷ ○
    $k \leftarrow \text{NextSubKey}(k, r)$
    for $i = 0$ to 15 do                                ▷ SB/SR: leak ×16b
        $s_i \leftarrow S'[s_i]$                         ▷ ●
        $s'_i \leftarrow s_{\rho(i)}$
    if $r < 10$ then
        for $i = 0$ to 12 by 4 do                       ▷ MC: leak ×52b
            $t \leftarrow m_2 \oplus s'_i \oplus s'_{i+1} \oplus s'_{i+2} \oplus s'_{i+3}$   ▷ ○
            for $j = 0$ to 3 do
                $u \leftarrow m_2 \oplus s'_{i+j} \oplus s'_{i+(j+1 \bmod 4)}$   ▷ ○
                $u \leftarrow 2u \oplus m_3$             ▷ ○
                $s_{i+j} \leftarrow t \oplus s'_{i+j} \oplus u$   ▷ ○
for $i = 0$ to 15 do
    $\text{ciph}_i \leftarrow s'_i \oplus k_i \oplus m_1$   ▷ unmask ciph
return ciph
```

3.3.2 GF(16) Masking Scheme (`1MaskGF16`)

This masking scheme also uses the single-mask protection in a mask correction setting. The algorithm implementing this masking scheme is detailed in Alg. 4. It takes as input a 16-byte plaintext $(\text{msg}_i)_{(i=0..15)}$ and a 16-byte key $(\text{key}_i)_{(i=0..15)}$. The S-Box step is here implemented via an inversion in GF(16). To this end, a mapping from GF(256) to GF(16) and its inverse are used.We combine it with the affine part of SubBytes Aff and use two tables to implement these steps: mapGF16 and mapGF256. Four precomputed tables that operate in GF(16) are also needed:

$$T_{d_1} : ((x \oplus m), m) \mapsto x^2 \times p_0 \oplus m$$
$$T_{d_2} : ((x \oplus m), (y \oplus m')) \mapsto ((x \oplus m) \oplus (y \oplus m')) \times (y \oplus m')$$
$$T_m : ((x \oplus m), (y \oplus m')) \mapsto ((x \oplus m) \times (y \oplus m'))$$
$$T_{inv} : ((x \oplus m), m) \mapsto x^{-1} \oplus m$$

where p_0 is the constant that defines the composite field arithmetic.

Here again, the initialization step consists in picking at random two masks. The first one, m_1, is the main mask. It is used to mask the input state, and to mask the inputs and outputs of the S-Box. A second mask m_2 is needed for the same reason as in the previous scheme: it is used temporarily in MixColumns to avoid unmasking. Some additional values are computed: $m_3 = m_2 \oplus 2m_2$, $m_4 = m_1 \oplus \mathrm{Aff}(m_1)$, $(m_l, m_h) = \mathrm{mapGF16}(m_1)$, $m_{hl} = m_h \oplus m_l$, $y = T_m[m_h, m_l]$ and $z = T_m[m_{hl}, m_l]$. The mask m_3 is used as before at the end of MixColumns to recover values masked with m_1. The other values are used in the computation of the non-linear part of SubBytes, here again to recover values masked with m_1. The computation of the inverse potentially leaks 13 nibbles in addition to the resulting byte, amounting to 208 leakages on nibbles and 16 leakages on bytes per round.

The algebraic modelling for this scheme is the same as for Alg. 3 for the MixColumns step. The SubBytes step requires some extra variables to represent the leakage equations. There are 208×4 extra bits that are linked to the other ones with the straightforward algebraic relation on the bits.

3.3.3 GF(16) Full Masking Scheme (`16MaskGF16`)

This masking scheme is very similar to the previous one. The only difference is that the state is now masked with a 128 bit mask. The algorithm is very similar to Alg. 4. We refer to this description for the leakage points.

The algebraic modelling of this masking scheme is identical to the `1MaskGF16` masking scheme, except that it requires 128 extra bits to represent the mask.

Algorithm 4: Operations and leakages for the `1MaskGF16` version of AES.

```
$m_1 \leftarrow$ rand                                   $\triangleright$ mask 1
$m_2 \leftarrow$ rand                                   $\triangleright$ mask 2
$m_3 \leftarrow m_2 \oplus 2m_2$                        $\triangleright$ mask correction MC
$m_4 \leftarrow m_1 \oplus \mathrm{Aff}(m_1)$           $\triangleright$ mask correction SB
$(m_l, m_h) \leftarrow \mathrm{mapGF16}(m_1)$           $\triangleright$ mask mapping
$m_{hl} \leftarrow m_h \oplus m_l$                      $\triangleright$ mask correction SB
$y \leftarrow T_m[m_h, m_l]$                            $\triangleright$ mask correction SB
$z \leftarrow T_m[m_{hl}, m_l]$                         $\triangleright$ mask correction SB

**for** $i = 0$ **to** 15 **do**
    $s_i \leftarrow \mathrm{msg}_i \oplus m_1$          $\triangleright$ mask msg
    $k_i \leftarrow \mathrm{key}_i$
**for** $r = 1$ **to** 10 **do**                        $\triangleright$ Rounds main loop
    **for** $i = 0$ **to** 15 **do**                    $\triangleright$ ARK: leak $\times$16b
        $s_i \leftarrow s_i \oplus k_i$                 $\triangleright$ ○
    $k \leftarrow \mathrm{NextSubKey}(k, r)$
    **for** $i = 0$ **to** 15 **do**   $\triangleright$ SB/SR: leak $\times$208n, $\times$16b
        $(\widetilde{a}_l, \widetilde{a}_h) \leftarrow \mathrm{mapGF16}(s_i)$
        $w \leftarrow T_m[\widetilde{a}_l, m_h]$        $\triangleright$ ●
        $x \leftarrow T_m[\widetilde{a}_h, m_l]$        $\triangleright$ ●
        $u_d \leftarrow T_{d1}[\widetilde{a}_h, m_h]$   $\triangleright$ ●
        $v_d \leftarrow T_{d2}[\widetilde{a}_h, \widetilde{a}_l]$   $\triangleright$ ●
        $f_d \leftarrow u_d \oplus v_d \oplus w \oplus x \oplus z$   $\triangleright$ ○
        $f'_d \leftarrow T_{\mathrm{inv}}[f_d, m_h]$    $\triangleright$ ●
        $f''_d \leftarrow f'_d \oplus m_{hl}, m$        $\triangleright$ ○
        $u_h \leftarrow T_m[f''_d, \widetilde{a}_h]$    $\triangleright$ ●
        $v_h \leftarrow T_m[f''_d, m_h]$                $\triangleright$ ●
        $f_{a_h} \leftarrow u_h \oplus m_h \oplus v_h \oplus x \oplus y$   $\triangleright$ ○
        $u_l \leftarrow T_m[f'_d, \widetilde{a}_l]$     $\triangleright$ ●
        $v_l \leftarrow T_m[f'_d, m_l]$                 $\triangleright$ ●
        $f_{a_l} \leftarrow u_l \oplus m_l \oplus v_l \oplus w \oplus f_{a_h} \oplus m_h \oplus y$   $\triangleright$ ○
        $s_i \leftarrow \mathrm{mapGF256}(\widetilde{a}_l, \widetilde{a}_h) \oplus m_4$   $\triangleright$ ●
        $s'_i \leftarrow s_{\rho(i)}$
    **if** $r < 10$ **then**
        **for** $i = 0$ **to** 12 **by** 4 **do**       $\triangleright$ MC: leak $\times$52b
            $t \leftarrow m_2 \oplus s'_i \oplus s'_{i+1} \oplus s'_{i+2} \oplus s'_{i+3}$   $\triangleright$ ○
            **for** $j = 0$ **to** 3 **do**
                $u \leftarrow m_2 \oplus s'_{i+j} \oplus s'_{i+(j+1 \bmod 4)}$   $\triangleright$ ○
                $u \leftarrow 2u \oplus m_3$            $\triangleright$ ○
                $s_{i+j} \leftarrow t \oplus s'_{i+j} \oplus u$   $\triangleright$ ○
**for** $i = 0$ **to** 15 **do**
    $\mathrm{ciph}_i \leftarrow s'_i \oplus k_i \oplus m_1$
**return** ciph
```

3.4 Software Framework

Generally speaking, the CNF format only use AND and OR operations. When modelling cryptographic function, the XOR operation appears very often. Converting an XOR operation into CNF leads to an increased number of clauses which can be blocking for the solver. In our software framework, we chose the `CryptoMiniSat` (Soos et al., 2009) solver (version 2.9.6) it has been especially designed to handle so-called XOR-clauses. XOR-clauses are an extension to the CNF format that is efficiently processed by `CryptoMiniSat`.

To perform the experiments, we have developed a full software framework which allows to

- Generate the equations for the AES cipher in algebraic form using Magma (Bosma et al., 1997), and convert those equations into CNF format with XOR clauses.
- Generate extra equations for newly introduced variables (e.g. the masks), for each implementation (i.e. each masking scheme). These equations have to bind the new variables to the cipher equations.
- Generate the equations for the leakages of the corresponding intermediate values (e.g. $\mathrm{HW}(a) = h$),

for each instance (i.e. key/message pair).

- Run the solver with selected equations for a given number of leakages. The leakages position can also be specified.
- Record the timing and the result of the solver (correct key found, incorrect key found, time-out/error). We set the timeout to 2 hours.

Our framework allows to generate several randomly generated instances to study the variance of the solving times.

4 GENERAL RESULTS

We applied ASCA to unprotected AES (`plain` implementation) to validate that we can reproduce the results found in the state of the art. We describe in this section our experiments on `plain` AES.

In this section and the next one, results are given in tables with the following notations: the first column shows the rounds that were observed, the second one gives the number of leakages. The column "Finish" gives the proportion of executions that finished before the time-out, and the column "Rate" gives the proportion of finished executions that output the right solution. The symbol $\star$ indicates that equations enforcing a ciphertext value have been added to the system (known ciphertext case).

4.1 Leakages from Key Scheduling

In (Renauld and Standaert, 2009), the authors state that if the key scheduling leaks information, the attack of (Mangard, 2003) can be applied directly with 81 leaking bytes. Still, we tried the ASCA approach on the key scheduling leakages to see whether ASCA can perform better than Mangard's approach. The results are given in Table 1.

Table 1: Results on Key Scheduling leakages.

Rnds	Leakages	Finish	Rate
1–4	64	100%	0%
1–5	80	100%	100%
1–4 $\star$	64	100%	100%
1–5 $\star$	80	100%	100%

We see that only 64 leaking bytes are required for the attack to succeed instead of 81 when a plaintext/ciphertext pair is available. The ASCA approach allows to recover the key more efficiently by automatizing the solving task.

In implementations secure against DSCA, as the key scheduling step does not involve any input data, it is not necessary to add masking. Thus, this attack applies directly to masked implementations. Though, as noted in (Renauld and Standaert, 2009), the key scheduling could have been pre-computed, and no leakages are available from this step. That is why, in the next sections, we will consider that no leakage from the key scheduling is available in our measurements.

4.2 Unprotected AES

For sake of completeness, we tried our framework on an unprotected AES implementation. A summary of the results obtained is given by Table 2. For MC, we consider two different settings: a first one where only the 16 results of MC leak, and another one where each intermediate result leaks, giving in this case a total of 52 leakages.

First, we tried the attack by feeding the leakages of the output of SB only. Indeed, template attacks work by profiling specific operations. Each time a different operation is considered, new templates must be constructed (and possibly, new curves have to be observed). As the look-up table of SB might be more visible than other operations because of the repeated memory accesses, assuming the attacker has only profiled this table access makes sense. The first two lines of Table 2 show that this attack fails, whether the ciphertext is available or not.

Next, we aimed at finding which is the minimal number of leakages required for the attack. We found out that the attack works with good probability when it is given 48 leakages from the first round and the ciphertext. This is better than the results by (Renauld et al., 2009). If we add the full MC leakage, then the attack works with probability one.

When the ciphertext is not available, the attack succeeds with good probability with 84 leakages from the first round, and with probability 1 with 96 leakages on two rounds.

Table 2: Results of the attack on an unprotected implementation.

Rnds	Leakages		Finish	Rate
	ARK,SB,MC	Total		
1	16,16,16	48	100%	0%
1	16,16,52	84	100%	62.5%
1–2	16,16,16	96	100%	100%
1–2	16,16,52	168	100%	100%
1 $\star$	16,16,16	48	100%	100%
1 $\star$	16,16,52	84	100%	100%
1–2 $\star$	16,16,16	96	100%	100%
1–2 $\star$	16,16,52	168	100%	100%

5 MASKED IMPLEMENTATIONS

5.1 Partially Masked Implementation

The first attack that we performed is on a partially masked implementation. As already said, leakages from the middle rounds are harder to exploit with DSCA and it is commonly admitted that DSCA can be efficiently prevented by masking only the first three and the last four rounds of AES. It is particularly interesting for constrained devices, as mentioned in (Akkar et al., 2004), (Akkar and Goubin, 2003) or (Tillich et al., 2007).

In this setting, the `plain` modelling can be used to attack such implementations, by using only the leakages from rounds 4 to 6.

Table 3 summarizes the results obtained with leakages from the middle rounds of AES with the `plain` implementation. Results show that a reasonable success rate can be achieved with the observation of only rounds 4 and 5, when MC leaks all intermediate values and the ciphertext is available. Full success rate is achieved as soon as leakages from rounds 4, 5 and 6 can be observed, even if MC leaks only 16 bytes and the ciphertext is unavailable.

It is interesting to note that when using middle rounds leakages, adding the information from the ciphertext is not only unnecessary, it also makes the solving time longer. This caused timeout in our examples.

Table 3: Results of the attack on partially masked implementations.

Rnds	Leakages ARK,SB,MC	Total	Finish	Rate
4,5	16,16,16	96	100%	100%
5,6	16,16,16	96	100%	100%
4,5,6	16,16,16	144	100%	100%
4,5	16,16,52	168	100%	100%
5,6	16,16,52	168	87.5%	100%
4,5,6	16,16,52	252	100%	100%
4,5 ⋆	16,16,16	96	0%	t/o
5,6 ⋆	16,16,16	96	0%	t/o
4,5,6 ⋆	16,16,16	144	100%	100%
4,5 ⋆	16,16,52	168	37.5%	100%
5,6 ⋆	16,16,52	168	100%	100%
4,5,6 ⋆	16,16,52	252	100%	100%

5.2 One Byte Masking Scheme

This section summarizes the results of the attacks on the `1Mask` version of AES, as described in Section 3.3.1. As described in Section 3, the modelling adds 16 new variables for the mask. We expected the system to require more leakages than the `plain` implementation to be solvable. We experimented several variants of the attack, their results are summarized in Table 4.

Results show that in some cases, the attack would succeed with only 48 first round leakages if the ciphertext is known. This is quite rare, we cannot rely on this in a single trace/message scenario, but it is worth mentioning that if the attacker has access to several traces, only 48 leakages may lead to a key recovery.

With unknown ciphertext, the solver will find a key candidate that fits the constraints from the first rounds, without being the expected key.

We reach a good success probability with either leakages from two consecutive rounds, or leakages from one round including leakages from MC intermediate values.

The results are similar when we add the leakages of the third round, but the execution takes longer. At this point, the more leakages we add, the longer take the execution, until it eventually reaches the timeout. This is because adding too many information forces the system to solve more equations, whilst it has enough information too find the right solution.

Table 4: Results of the attack on `1Mask` implementation.

Rnds	Leakages ARK,SB,MC	Total	Finish	Rate
1	16,16,16	48	100%	0%
1	16,16,52	84	62.5%	100%
1–2	16,16,16	96	25.0%	100%
1–3	16,16,16	144	37.5%	100%
1–4	16,16,16	192	0%	t/o
1 ⋆	16,16,16	48	12.5%	100%
1 ⋆	16,16,52	84	87.5%	100%
1–2 ⋆	16,16,16	96	62.5%	100%
1–3 ⋆	16,16,16	144	62.5%	100%
1–4 ⋆	16,16,16	192	75.0%	100%

5.3 GF(16) One Byte Masking Scheme

This section summarizes the results of the attacks on the `1MaskGF16` version of AES, as described in Section 3.3.2.

In this implementation, during the S-Box computation, most of the operations are performed on 4-bit nibbles, especially table-lookups. Knowing the Hamming weight of a nibble gives a lot of information. Thus we expected the attacks on such implementation to be easier with ASCA.

In our experiments summarized in Table 5, we focused on leakages coming from table-lookups only. This is suitable for situations where the profiling phase is done only for these operations.

We first tried to minimize the number of necessary leakages by exploiting the leakages from the 4-bit table-lookups in the order of the algorithm (see Alg. 4), and the final 8-bit table lookup. First lines show that 64 leakages from the first round are enough, provided that the ciphertext is available and used. Otherwise, the system is under-defined, and the SAT solver finishes its computation with erroneous solutions.

The attack succeeds without the ciphertext with as little as 160 table access leakages in the first round. We can conclude that this masking scheme is very weak against ASCA due to the manipulation of 4-bit values.

Table 5: Results of the attack on `1MaskGF16` implementation.

Rnds	Leakages		Finish	Rate
	ARK,SB,MC	Total		
1	0,48,0	48	100%	0%
1	0,64,0	64	100%	0%
1	0,80,0	80	100%	12.5%
1	0,96,0	96	100%	37.5%
1	0,112,0	112	100%	75.0%
1	0,128,0	128	100%	100%
1	0,144,0	144	100%	100%
1	0,160,0	160	100%	100%
1 ⋆	0,48,0	48	0%	t/o
1 ⋆	0,64,0	64	100%	100%
1 ⋆	0,80,0	80	100%	100%
1 ⋆	0,96,0	96	100%	100%
1 ⋆	0,112,0	112	100%	100%
1 ⋆	0,128,0	128	100%	100%
1 ⋆	0,144,0	144	100%	100%
1 ⋆	0,160,0	160	100%	100%

5.4 GF(16) Full Masking Scheme

This section summarizes the results of the attacks on the `16MaskGF16` version of AES, as described in Section 3.3.3. For the same reasons as `1MaskGF16`, we chose to focus only on table lookups leakages for this implementation. The results are summarized in Table 6.

Attacking this masking scheme is significantly more difficult than the one with only 1 mask: with known ciphertext, the attack is not possible with less than 128 leakages. Finally, 160 leakages are not enough anymore to reach the 100% success rate with unknown plaintext. Regarding the known plaintext case, the same amount of leakages is more difficult to handle, resulting in more unfinished computations.

Table 6: Results of the attack on `16MaskGF16` implementation.

Rnds	Leakages		Finish	Rate
	ARK,SB,MC	Total		
1	0,48,0	48	100%	0%
1	0,64,0	64	100%	0%
1	0,80,0	80	100%	0%
1	0,96,0	96	100%	0%
1	0,112,0	112	100%	0%
1	0,128,0	128	100%	0%
1	0,144,0	144	100%	12.5%
1	0,160,0	160	100%	79.1%
1–2	0,160,0	320	100%	100%
1 ⋆	0,48,0	48	0%	t/o
1 ⋆	0,64,0	64	0%	t/o
1 ⋆	0,80,0	80	0%	t/o
1 ⋆	0,96,0	96	0%	t/o
1 ⋆	0,112,0	112	0%	t/o
1 ⋆	0,128,0	128	12.5%	100%
1 ⋆	0,144,0	144	87.0%	100%
1 ⋆	0,160,0	160	75.0%	100%
1–2 ⋆	0,160,0	320	100%	100%

6 CONCLUSION

In this work, we adopted an experimental approach to test the resistance of masked implementations against ASCA. Our approach is slightly different than in other papers as we focused on finding the minimal number of leakages required to break a given implementation, and we limited leakage types to suit the profiling stage. Our work not only confirms the results provided in previous works, but also gives more experimental evidences that the choice of the masking scheme has a direct impact on the feasibility of ASCA, even if those masking schemes have the same level of security regarding DSCA.

Our experiments confirm the intuition that the more variables are necessary to model the encryption, the more difficult the solving step will be. In particular, this makes masking scheme that use a 128-bit mask quite resistant to ASCA. Our experiments also highlight that the Hamming weight leakages on nibbles gives a lot of exploitable information for ASCA, making the GF(16) based masking schemes an easy target.

REFERENCES

Akkar, M.-L., Bevan, R., and Goubin, L. (2004). Two power analysis attacks against one-mask methods. In Roy, B. K. and Meier, W., editors, *FSE 2004*, volume 3017 of *LNCS*, pages 332–347. Springer.

Akkar, M.-L. and Giraud, C. (2001). An implementation of DES and AES, secure against some attacks. In Koç, Çetin Kaya., Naccache, D., and Paar, C., editors, *CHES 2001*, volume 2162 of *LNCS*, pages 309–318. Springer.

Akkar, M.-L. and Goubin, L. (2003). A generic protection against high-order differential power analysis. In Johansson, T., editor, *FSE 2003*, volume 2887 of *LNCS*, pages 192–205. Springer.

Banciu, V. and Oswald, E. (2014). Pragmatism vs. elegance: comparing two approaches to simple power attacks on AES. Cryptology ePrint Archive, Report 2014/177.

Banciu, V., Oswald, E., and Whitnall, C. (2015). Reliable information extraction for single trace attacks. In Nebel, W. and Atienza, D., editors, *DATE 2015*, pages 133–138. ACM.

Bard, G. V., Courtois, N. T., and Jefferson., C. (2007). Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-Solvers. Cryptology ePrint Archive, Report 2007/024.

Biryukov, A. and De Cannière, C. (2003). Block ciphers and systems of quadratic equations. In Johansson, T., editor, *FSE 2003*, volume 2887 of *LNCS*, pages 274–289. Springer.

Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J. B., Seurin, Y., and Vikkelsoe, C. (2007). PRESENT: An ultra-lightweight block cipher. In Paillier, P. and Verbauwhede, I., editors, *CHES 2007*, volume 4727 of *LNCS*, pages 450–466. Springer.

Bosma, W., Cannon, J., and Playoust, C. (1997). The Magma algebra system I: The user language. *J. Symb. Comput.*, 24(3-4):235–265.

Carlet, C., Faugère, J., Goyet, C., and Renault, G. (2012). Analysis of the algebraic side channel attack. *J. Cryptographic Engineering*, 2(1):45–62.

Chari, S., Jutla, C. S., Rao, J. R., and Rohatgi, P. (1999). Towards sound approaches to counteract power-analysis attacks. In Wiener, M. J., editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer.

Chari, S., Rao, J. R., and Rohatgi, P. (2003). Template attacks. In Kaliski Jr., B. S., Koç, Çetin Kaya., and Paar, C., editors, *CHES 2002*, volume 2523 of *LNCS*, pages 13–28. Springer.

Courtois, N. and Bard, G. V. (2007). Algebraic cryptanalysis of the data encryption standard. In Galbraith, S. D., editor, *Cryptography and Coding*, volume 4887 of *LNCS*, pages 152–169. Springer.

Courtois, N. and Pieprzyk, J. (2002). Cryptanalysis of block ciphers with overdefined systems of equations. In Zheng, Y., editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 267–287. Springer.

Genelle, L., Prouff, E., and Quisquater, M. (2010). Secure multiplicative masking of power functions. In Zhou, J. and Yung, M., editors, *ACNS 10*, volume 6123 of *LNCS*, pages 200–217. Springer.

Golic, J. D. and Tymen, C. (2003). Multiplicative masking and power analysis of AES. In Kaliski Jr., B. S., Koç, Çetin Kaya., and Paar, C., editors, *CHES 2002*, volume 2523 of *LNCS*, pages 198–212. Springer.

Goubin, L. and Patarin, J. (1999). DES and differential power analysis (the "duplication" method). In Koç, Çetin Kaya. and Paar, C., editors, *CHES'99*, volume 1717 of *LNCS*, pages 158–172. Springer.

Goudarzi, D. and Rivain, M. (2016). On the multiplicative complexity of boolean functions and bitsliced higher-order masking. In Gierlichs, B. and Poschmann, A. Y., editors, *CHES 2016*, volume 9813 of *LNCS*, pages 457–478. Springer.

Grosso, V. and Standaert, F.-X. (2015). ASCA, SASCA and DPA with enumeration: Which one beats the other and when? In Iwata, T. and Cheon, J. H., editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 291–312. Springer.

Herbst, C., Oswald, E., and Mangard, S. (2006). An AES smart card implementation resistant to power analysis attacks. In Zhou, J., Yung, M., and Bao, F., editors, *ACNS 06*, volume 3989 of *LNCS*, pages 239–252. Springer.

Heuser, A. and Zohner, M. (2012). Intelligent machine homicide - breaking cryptographic devices using support vector machines, COSADE, 2012. In (Schindler and Huss, 2012), pages 249–264.

Ishai, Y., Sahai, A., and Wagner, D. (2003). Private circuits: Securing hardware against probing attacks. In Boneh, D., editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer.

Mangard, S. (2003). A simple power-analysis (spa) attack on implementations of the AES key expansion. In Lee, P. J. and Lim, C. H., editors, *ICISC 02*, volume 2587 of *LNCS*, pages 343–358. Springer.

Messerges, T. S. (2001). Securing the AES finalists against power analysis attacks. In Schneier, B., editor, *FSE 2000*, volume 1978 of *LNCS*, pages 150–164. Springer.

Mohamed, M. S. E., Bulygin, S., Zohner, M., Heuser, A., and Walter, M. (2012). Improved algebraic side-channel attack on AES. Cryptology ePrint Archive, Report 2012/084.

Oren, Y., Kirschbaum, M., Popp, T., and Wool, A. (2010). Algebraic side-channel analysis in the presence of errors. In Mangard, S. and Standaert, F.-X., editors, *CHES 2010*, volume 6225 of *LNCS*, pages 428–442. Springer.

Oren, Y., Renauld, M., Standaert, F.-X., and Wool, A. (2012). Algebraic side-channel attacks beyond the hamming weight leakage model. In Prouff, E. and Schaumont, P., editors, *CHES 2012*, volume 7428 of *LNCS*, pages 140–154. Springer.

Oren, Y. and Wool, A. (2012). Tolerant algebraic side-channel analysis of AES. Cryptology ePrint Archive, Report 2012/092.

Oswald, E., Mangard, S., and Pramstaller, N. (2004). Secure and efficient masking of AES - a mission impossible? Cryptology ePrint Archive, Report 2004/134.

Oswald, E., Mangard, S., Pramstaller, N., and Rijmen, V. (2005). A side-channel analysis resistant description of the AES S-box. In Gilbert, H. and Handschuh, H., editors, *FSE 2005*, volume 3557 of *LNCS*, pages 413–423. Springer.

Oswald, E. and Schramm, K. (2006). An efficient masking scheme for AES software implementations. In Song, J., Kwon, T., and Yung, M., editors, *WISA 05*, volume 3786 of *LNCS*, pages 292–305. Springer.

Prouff, E. and Rivain, M. (2008). A generic method for secure SBox implementation. In Kim, S., Yung, M., and Lee, H.-W., editors, *WISA 07*, volume 4867 of *LNCS*, pages 227–244. Springer.

Rebeiro, C., Selvakumar, A. D., and Devi, A. S. L. (2006). Bitslice implementation of AES. In Pointcheval, D., Mu, Y., and Chen, K., editors, *CANS 06*, volume 4301 of *LNCS*, pages 203–212. Springer.

Renauld, M. and Standaert, F.-X. (2009). Algebraic side-channel attacks. Cryptology ePrint Archive, Report 2009/279.

Renauld, M., Standaert, F.-X., and Veyrat-Charvillon, N. (2009). Algebraic side-channel attacks on the AES: Why time also matters in DPA. In Clavier, C. and Gaj, K., editors, *CHES 2009*, volume 5747 of *LNCS*, pages 97–111. Springer.

Schindler, W. and Huss, S. A., editors (2012). *COSADE 2012*, volume 7275 of *LNCS*. Springer.

Song, L., Hu, L., Sun, S., Zhang, Z., Shi, D., and Hao, R. (2014). Error-tolerant algebraic side-channel attacks using BEE. Cryptology ePrint Archive, Report 2014/683.

Soos, M., Nohl, K., and Castelluccia, C. (2009). Extending SAT solvers to cryptographic problems. In Kullmann, O., editor, *SAT 2009*, volume 5584 of *LNCS*, pages 244–257. Springer.

Tillich, S., Herbst, C., and Mangard, S. (2007). Protecting AES software implementations on 32-bit processors against power analysis. In Katz, J. and Yung, M., editors, *ACNS 07*, volume 4521 of *LNCS*, pages 141–157. Springer.

Trichina, E., De Seta, D., and Germani, L. (2003). Simplified adaptive multiplicative masking for AES. In Kaliski Jr., B. S., Koç, Çetin Kaya., and Paar, C., editors, *CHES 2002*, volume 2523 of *LNCS*, pages 187–197. Springer.

Veyrat-Charvillon, N., Gérard, B., and Standaert, F.-X. (2014). Soft analytical side-channel attacks. In Sarkar, P. and Iwata, T., editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 282–296. Springer.

Zhao, X., Zhang, F., Guo, S., Wang, T., Shi, Z., Liu, H., and Ji, K. (2012). MDASCA: an enhanced algebraic side-channel attack for error tolerance and new leakage model exploitation, COSADE, 2012. In (Schindler and Huss, 2012), pages 231–248.