COMBINATIONAL LOGIC DESIGN FOR AES SUBBYTE TRANSFORMATION ON MASKED DATA

ELENA TRICHINA

Abstract. Low power consumption, low gate count and high throughput used to be standard design criteria for cryptographic coprocessors designated for smart cards and related embedded devices. Not anymore. With the advent of side channel attacks, the first and foremost concern is device resistance to such attacks, at a price.

General-purpose hardware countermeasures, such as metal shields, tamper-detectors, clock randomization, random current generators, seem not anymore a sufficient protection against sophisticated differential power analysis or fault attacks specifically tailored to break a particular device. In this paper we describe an attempt to embed data masking technique at a hardware design level for an AES coprocessor. We concentrate on inversion in GF since it is the only non-linear operation, and requires complex transformations on the masked data and masks.

1. Introduction

When first invasive attacks on smart cards had been reported in mid-90ies [2], chip industry reacted by incorporating in a chip a plethora of tamper resistant features such as glue logic, metal shields, various sensors, etc. [14]

The end of the 90ies sent once again a shock wave through the industry. A new class of so called *side-channel* attacks emerged as a powerful thread to cryptographic applications, and ever since there is a steady stream of publications divulging how to break various implementations of DES, AES and RSA by measuring timing characteristics [12], power consumption [11, 17, 18] or electromagnetic radiation [9, 22] of a smart card microprocessor when it runs the algorithm in question.

Until recently, most of the attacks exploited some specific features of software implementation of cryptographic algorithms. Fittingly, most countermeasures against such attacks were also designed at algorithmic level, although often it was assumed that hardware features such as clock randomization or random current generation would make an attack much more difficult, and thus, less practical.

For many applications, however, it is necessary that cryptographic algorithm should be realized in hardware. Hence, research into the vulnerability of cryptographic hardware is just as important. Although not many results had been published yet, it is prudent to suggest that cryptographic hardware also leaks side channel information, and that alongside with general tamper-resistant features such hardware should include countermeasures specifically targeted to protect the realized algorithm(s).

Key words and phrases. AES, DPA attack, composite field, inversion in GF, data masking.

In this paper we propose rather a novel approach to protecting AES coprocessor from differential power analysis attacks, namely full hardware implementation of AES computations on *masked data*.

The main innovation is that we suggest a new method of computing inverses on masked data in composite fields. As one may surmise, this comes at a price in terms of gate count and power consumption. To minimize these parameters, we offer yet another, less expensive, solution that combines ideas of inversion in composite fields [24, 29, 19] with masked table look-up [28].

The rest of the paper is organized as follows. After a brief description of the Advanced Encryption Standard algorithm in the next chapter, we proceed in Chapter 3 explaining in details how to reduce inversion in the field $GF(2^8)$ to inversion in the composite field $GF((2^4)^2)$, and how the latter can be realized in combinational logic only.

Chapter 4 introduces the notion and discusses the difficulties of the inversion on masked data, i.e., data that are obtained as a result of the XOR operation applied to the actual data $A_{i,j}$ constituting an (i,j)-th byte of the state, and a random mask $X_{i,j}$.

In the subsequent chapter we outline hardware blocks that implement the operations of mapping from $GF(2^8)$ to $GF((2^4)^2)$ and inversion in $GF(2^4)$ on masked data, and estimate the cost in terms of a gate count.

Chapter 6 describes an alternative solution to hardware implementation of inversion on masked data in $GF(2^4)$ using masked lookup tables.

The paper is concluded with the the summary of the novel features for secure AES hardware architecture and with discussion on price/performance tradeoffs.

2. AES Reminder

AES encryption and decryption are based on four different transformations that are performed repeatedly in a certain sequence; each transformation maps a 128-bit input state into a 128-bit output state. In both states, transformations are grouped in rounds. The rounds are slightly different for encryption and decryption, and the number of rounds depends on the key size.

For simplicity, we consider the 128-bit block- and and key sizes version on the basis that the cryptanalytic study of the Rijndael during the standardization process was primarily focused on this version. For a complete mathematical specification of the Rijndael algorithm we refer readers to [7].

In the Rijndael, the 128-bit data block is considered as a 4×4 array of bytes. The algorithm consists of an initial data/key addition, 9 full rounds (when the key length is 128 bits), and a final (modified) round. A separate key scheduling module is used to generate all the sub-keys, or *round keys*, from the initial key; a sub-key is also represented as 4×4 array of bytes. The full Rijndael round involves four steps.

The *Byte Substitution* step replaces each byte in a block by its substitute in an S-box. The S-box is an invertible substitution table which is constructed by a composition of two transformations:

- First, each byte $A_{i,j}$ is replaced with its reciprocal in $GF(2^8)$ (except that 0, which has no reciprocal, is replaced by itself).
- Then, an affine transformation f is applied. It consists of
  - a bitwise matrix multiply with a fixed 8×8 binary matrix M,

– after which the resultant byte is XOR-ed with the hexadecimal number {63}.

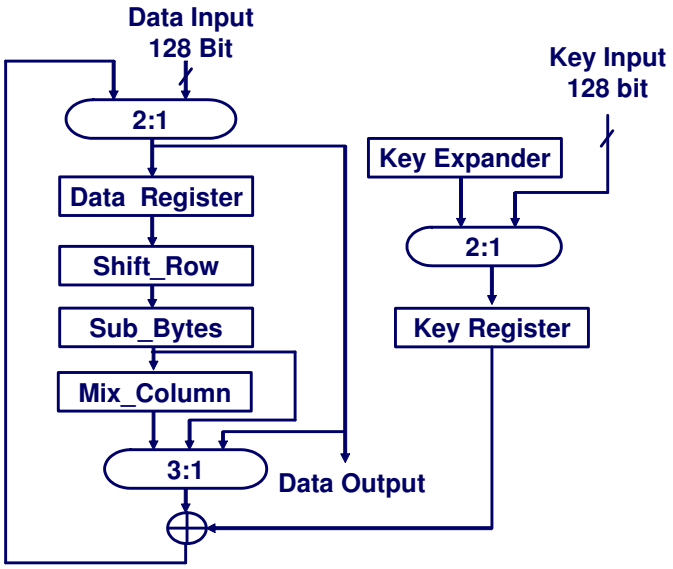

FIGURE 1. The structure of the AES encryption algorithm.

The S-box is usually implemented as a look-up table consisting of 256 entries; each entry is 8 bits wide; but it also can be computed "on-a-fly".

Next comes the *Shift Row* step. Each row in a 4×4 array of bytes of data is shifted 0, 1, 2 or 3 bytes to the left in a round fashion, producing a new 4×4 array of bytes.

In the *Mix Column* step, each column in the resultant 4×4 array of bytes is considered as polynomial over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = \{03\}x^3 + \{01\}x^2 +' \{01\}x + \{02\}$. The operation of a multiplication with a fixed polynomial $a(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0$ can be written as a matrix multiplication where the matrix is a circular matrix with the first row equal to a_0, a_3, a_2, a_1, each subsequent row is obtained by a circular shift of the previous one by 1 position to the left. representation.

The final step, *Add Round Key*, simply XOR-es the result with the sub-key for the current round.

In parallel to the round operation, the round key is computed in the *Key Scheduling Block*. The round key is derived from the cipher key by means of key expansion and round key selection.

Round keys are taken from the expanded key (which is a linear array of 4-byte words) in the following way: the first round key consists of the first N_b words, the second of the following N_b words, etc. The first N_k words are filled in with the cipher key. Every following word $W[i]$ is obtained by XOR-ing the words $W[i-1]$ and $W[i-N_k]$.

For words in positions that are multiples of N_k, the word is first rotated by one byte to the left; then its bytes are transformed using the S-box from the *Byte Substitution* step, after which XOR-ed with the round-dependent constant.

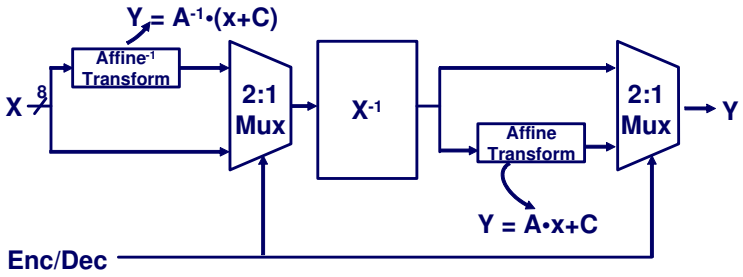

FIGURE 2. Two steps of Byte Substitution transformation.

Fig. 1 shows a standard circuit implementation of the AES encryption algorithm, which executes one round per once clock. A sequence of round operations is implemented as a combinational circuit and its input and output are connected to a 128-bit data register. The 3:1 selector before the *AddRoundKey*is used to skip some operations in the first and the last rounds.

Compared to the encryption, the decryption algorithm is simply the execution of the inverse transformations in the inverse order.

In applications such as smart cards, hardware complexity is a very important issue that directly affects the cost and power consumption. In general, one may claim that the size, the speed and the power consumption of an AES hardware engine depends to a great degree on the number and the style of implementation of S-boxes.

To optimize silicon area, a number of flexible ASIC solutions were proposed [15, 16] that use similarities between encryption and decryption to share silicon. For this, it is necessary to implement *SubByte* transformation in two separate steps, as a combination of the inversion in the field and an affine transformation (or its inverse).

While the affine transformations used for encryption and decryption are slightly different, the silicon implementing an inversion in $GF(2^8)$ can be used for both. Therefore, area- and power-efficient and secure implementation of inversion in $GF(2^8)$ can have a big impact on overall design.

The most obvious solution is to use a look-up table for this operation. It is fast and inexpensive in terms of power consumption, as the study in [19] indicates. There is a major drawback, however. Namely, the size of required memory, which is about 1,700 gate equivalents per one table in 0.18μ technology.

Various alternative approaches to constructing compact inversion circuits over GF have been studied. In particular, composite field inversions were found to be effective over $GF(2^8)$, and were used to create compact AES implementations [20, 24, 25, 19, 29]. As a basis for our design we used fully combinational logic implementation of inversion in composite field described in [29].

In contrast to V. Rijmen original proposal [23] which suggests the optimal normal basis representation of finite field elements, [29] use the polynomial representation of finite field elements, which results in a far more flexible hardware architecture without necessity to do complex conversions from one representation to another.

3. Mathematical Background

First of all, we need some mathematical background. To make this paper self-contained but not overloaded with mathematics, we introduce only relevant formulae, which we borrow from [29]. Fig. 3 will help to relate the transformations defined below.

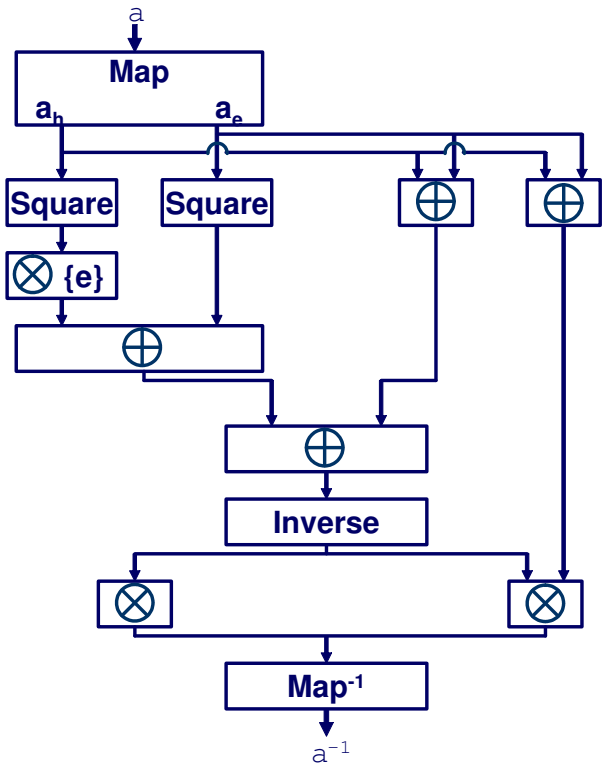

Figure 3. Implementation of inversion in composite fields.

3.1. **Transition between field representations.** Usually, the field $GF(2^8)$ is seen as an extension of $GF(2)$ and therefore its elements can be represented as bytes. However, $GF(2^8)$ can also bee seen as a quadratic extension of $GF(2^4)$; in this case an element $a \in GF(2^8)$ is represented as a linear polynomial $a_h x + a_l$ with coefficients in $GF(2^4)$. We denote is as $[a_h, a_l]$. This isomorphic representation is far better suited for hardware implementation [20, 24, 19, 29].

The bijection from an element $a = (a_0, a_1, ..., a_7)$ to a two-term polynomial $a_h x + a_l$ is given by the function *map* computed as shown below.

$$a_h x + a_l = map(a), a_h, a_l \in GF(2^4), a \in GF(2^8)$$

$$\begin{array}{lllll} & a_A = a_1 \oplus a_7 & a_B = a_5 \oplus a_7 & a_C = a_4 \oplus a_6 & \\ (3.1) & a_{l0} = a_C \oplus a_0 \oplus a_5 & a_{l1} = a_1 \oplus a_2 & a_l 2 = a_A & a_{l3} = a_2 \oplus a_4 \\ & a_{h0} = a_C \oplus a_5 & a_{h1} = a_A \oplus a_C & a_{h2} = a_B \oplus a_2 \oplus a_3 & a_{h3} = a_B \end{array}$$

The inverse transformation (map^{-1}) converts a two-term polynomial back into an element $a \in GF(2^8)$, and is defined by a similar set of equations, comprising only binary XOR, or $\oplus$, operations. For more details see [29].

3.2. **Arithmetic operations in extension fields.** All arithmetic operations applied to elements of $GF(2^8)$ can also be computed in the new representation.

Two-term polynomials are added by addition of their corresponding coefficients:

$$(a_h x + a_l) \oplus (b_h x + b_l) = (a_h \oplus b_h)x + (a_l \oplus b_l). \tag{3.2}$$

Multiplication and inversion of two term-polynomials require a modular reduction step to ensure that the result is a two-term polynomial as well. Following [29], we use irreducible polynomial $n(x) = x^2 = \{1\}x + \{e\}$. Coefficients of $n(x)$ are elements in $GF(2^4)$ are written in hexadecimal notation. Their particular values are chosen to optimize the finite field arithmetics.

A convenient method to multiply $a(x)$ and $b(x)$ in $GF(2^k)$ is to generate partial products: $P(x) = a(x) \cdot x^i$, and to add those partial products where corresponding bit of the multiplier is 1. The partial products can be calculated efficiently by iterating multiplication by x, i.e., $P_i(x) = P_{i-1}(x) \cdot x \mod m(x), P_0(x) = a(x)$. Multiplication by x is termed *xtimes*.

Multiplication of two-term polynomials involves multiplication of their coefficients, which are elements of $GF(2^4)$, which require an irreducible polynomial of degree 4 $m(x) = x^2 + x + 1$.

Multiplication in $GF(2^4)$ is given by the following set of equations derived by applying the formulae for polynomial multiplication to $GF(2^4)$ with $m(x)$ as irreducible polynomial.

$$q(x) = a(x) \otimes b(x) = a(x) \cdot b(x) \mod m_4(x), \text{where} a(x), b(x), q(x) \in GF(2^4) \tag{3.3}$$

$$\begin{array}{ll} q_0 = a_0 b_0 \oplus a_3 b_1 \oplus (a_2 \oplus a_3) b_2 \oplus (a_1 \oplus a_2) b_3 & q_1 = a_1 b_0 \oplus (a_0 \oplus a_3) b_1 \oplus a_2 b_2 \oplus a_1 b_3 \\ q_2 = a_2 b_0 \oplus a_1 b_1 \oplus (a_0 \oplus a_3) b_2 \oplus (a_2 \oplus a_3) b_3 & q_3 = a_3 b_0 \oplus a_2 b_1 \oplus a_1 b_2 \oplus (a_0 \oplus a_3) b_3 \end{array}$$

The concatenation of two bits $a_i a_j$ in these equations represents binary multiplication, which is an AND operation.

Squaring in $GF(2^4)$ is a special case of multiplication, and is computed as follows.

$$q(x) = a(x)^2 \mod m_4(x), a(x), q(x) \in GF(2^4)$$

$$q_0 = a_0 \oplus a_2 \quad q_1 = a_2 \quad q_2 = a_1 \oplus a_3 \quad q_3 = a_3 \tag{3.4}$$

Inversion of a two-term polynomial is the equivalent operation to inversion in $GF(2^8)$ defined as $(a_h x + a_l) \otimes (a_h x + a_l)^{-1} = \{0\}c + \{1\}$. From this definition the formulae for inversion can be derived:

$$(a_h x + a_l)^{-1} = (a_h \otimes d)x + (a_h \oplus a_l) \otimes d, d = ((a_h^2 \otimes \{e\}) \oplus (a_h \otimes a_l) \oplus a_l^2)^{-1}. \tag{3.5}$$

The inverse a^{-1} of an element $a \in GF(2^4)$ can be derived by solving the equation $a \otimes a^{-1} \mod m(x) = 1$ as follows.

$$q(x) = a(x)^{-1} \mod m_4(x), q(x), a(x) \in GF(2^4) \tag{3.6}$$

where $q = (q_0, .., q_3)$ is calculated as shown below with $a_A = a_1 \oplus a_2 \oplus a_3 \oplus a_1a_2a_3$.

$$\begin{aligned} q_0 &= a_A \oplus a_0 \oplus a_0a_2 \oplus a_1a_2 \oplus a_0a_1a_2 \\ q_1 &= a_0a_1 \oplus a_0a_2 \oplus a_1a_2 \oplus a_3 \oplus a_1a_3 \oplus a_0a_1a_3 \\ q_2 &= a_0a_1 \oplus a_2 \oplus a_0a_2 \oplus a_3 \oplus a_0a_1 \oplus a_0a_2a_3 \\ q_3 &= a_A \oplus a_0a_3 \oplus a_1a_3 \oplus a_2a_3 \end{aligned}$$

In contrast to inversion in $GF(2^8)$, inversion in $GF(2^4)$ is suitable for a hardware implementation using combinational logic.

4. Secure AES Implementation in Hardware: Computations on Masked Data

Basically, side-channel attacks work because there is a correlation between the physical measurements taken during computations (e.g., power consumption, EMF radiation, time of computations) and the internal state of the processing device, which itself is related to a secret key.

Among many attacks, the Differential Power Analysis (DPA) is the most dangerous (see, for example, [18]). It uses statistical analysis to extract information from a collection of power consumption curves obtained by running an algorithm many times with different inputs. Then an analysis of a probability distribution of point son the curves is carried on. DPA uses correlation between power consumption and specific key-dependent bits which appear at known steps of the cryptographic computations. For example, a selected bit b at the output of one S-box of the first round of AES will depend on the known input message and 8 unknown bits of the key. The correlation between power consumption and b can be computed for all 256 values of 8 unknown bits of the key. The correlation is likely to be maximal for the correct guess of the 8 bits of the key. Then an attack can be repeated for the remaining S-boxes.

There are many strategies to combat side-channel attacks. On the hardware level, the counter measures usually include clock randomization [26, 13, **?**], power consumption randomization [3] or compensation [8], and various detectors of abnormal behavior. However, as is already known within the smart card industry, the effect of some of these counter measures can be reduced by various signal processing techniques.

It seems that the most powerful counter measure is *bit splitting* [6, 10], which can be reduced to *masking* with random value. The idea is simple: the message (and the key) is masked by with some random mask at the beginning of the algorithm, and thereafter everything is almost as usual. Of course, the value of the mask at the end of some fixed step (e.g., at the end of the round) must be known in order to re-establish the expected value at the end of the execution; we call this *mask correction*.

A traditional XOR operation is used as a masking counter measure; however, the mask is arithmetic on $GF(2^8)$ [1]. The operation is compatible with the AES structure except for *SubByte*, which is the only non-linear transformation since it

uses an inversion in the field. In other words, it is easy to compute mask correction for all transformations in a round, apart from the inversion step of the *SubByte*.

Our solution is to modify the inversion in $GF(2^8)$ in such a way that it takes into account a mask. In other words, we define a new operation, *ModInv*, such that

$$ModInv(A_{i,j} \oplus X_{i,j}) = A_{i,j}^{-1} \oplus Y_{i,j}$$

for every byte $A_{i,j}$ of the state and some random mask $X_{i,j}$, and show how this operation can be implemented directly in combinational logic. Here $Y_{i,j}$ is either a new random mask, or is equal to an "old" mask $X_{i,j}$.

In the remaining part of the paper we show how to implement an inversion on masked data and compute a corresponding mask correction with combinational logic only, never revealing the actual data $A_{i,j}$ in a process.

4.1. **Operations on Masked Data.** As one can easily see, all operations in extension field as well as the function *map* eventually require only operations in $GF(2)$, namely bit-wide XOR and AND.

XOR is a linear operation, i.e., to compute XOR on two masked data bits, one does not need to "unmask" these bits:

$$(a_i \oplus x_i) \oplus (b_j \oplus y_j) = (a_j \oplus b_j) \oplus (x_j \oplus y_j).$$

Hence, a transformation of a masked byte $A_{i,j} \oplus X_{i,j}$ into a two-term polynomial which uses only bit-wide XOR (and the corresponding mask correction) can be carried out without revealing the actual data in a process, i.e., the following holds:

$$(map(A_{i,j} \oplus X_{i,j}) = map(A_{i,j}) \oplus map(X_{i,j}).$$

The same is applicable for the inverse transformation map^{-1}.

Since the affine transformation comprising the *SubByte* operation is a linear operation, and can be expressed only in terms of bit-wise XOR (see [29] for reference), it can be easily done on the masked data as well. The same applies to the inverse affine transformation used for decryption.

The only problematic part of the *SubByte* is an inversion, which, as we have seen, can be reduced to multiplication and inversion in $GF(2^4)$, both of them containing binary AND operation.

The problem is that in order to compute AND on masked data and the corresponding "mask correction", one would have to reveal the actual (unmasked) data bits. Indeed, if x_i and y_j denote the bits that mask the "real" bits a_i and b_j, then

$$(a_i \oplus x_i) \cdot (b_j \oplus y_j) = (a_i \cdot b_j) \oplus (a_i \cdot y_j) \oplus (x_i \cdot b_j) \oplus (x_i \cdot y_j).$$

Hence, basically, the problem of the inversion of masked data can be effectively reduced to the problem of computing a binary AND operation on masked bits of the data and the corresponding mask correction without revealing the actual (unmasked) data bits. In what follows, we show how this operation can be implemented in combinational logic without "compromising" the actual data bits.

4.2. **Masked AND operation.** Our approach is based on the algebraic properties of the operations $\oplus$ and $\cdot$. We derive the solution by simple manipulations with the algebraic formulas, as following. Let us denote masked bits a_i and b_j via $\tilde{a} = (a_i \oplus x_i)$ and $\tilde{b} = (b_j \oplus y_j)$ correspondingly. Then

$$\tilde{a} \cdot \tilde{b} = (a_i \cdot b_j \oplus (a_i \cdot y_j) \oplus (x_i \cdot b_j) \oplus (x_i \cdot y_j) \tag{4.1}$$

Now we want to express the terms $(a_i \oplus y_j)$ and $(b_j \oplus x_i)$ using only masked bits $\tilde{a}$, $\tilde{b}$ and the bits of the mask x_i and y_j. From the equation $x_i \cdot \tilde{b} = (x_i \cdot b_j) \oplus (x_i \cdot y_j)$ we can derive the method to compute $x_i \cdot b_j$ as follows.

$$(x_i \cdot b_j) = x_i \cdot \tilde{b} \oplus (x_i \cdot y_j) \tag{4.2}$$

Analogously, using equation $y_j \cdot \tilde{a} = (y_j \cdot a_i) \oplus (y_j \cdot x_i)$ we compute $a_i \cdot y_j$:

$$(a_i \cdot y_j) = y_j \cdot \tilde{a} \oplus (x_i \cdot y_j) \tag{4.3}$$

Now, substituting the corresponding terms in equation 4.1 for their values obtained in equations 4.2 and 4.3 and simplifying the resulting formulae, we obtain

$$\tilde{a} \cdot \tilde{b} = (a_i \cdot b_j) \oplus (x_i \cdot \tilde{b}) \oplus (y_j \cdot \tilde{a}) \oplus (x_i \cdot y_j) \tag{4.4}$$

Hence, the computations of the "mask correction" $(x_i \cdot \tilde{b}) \oplus (y_j \cdot \tilde{a}) \oplus (x_i \cdot y_j)$ can be carried out without compromising the bits of actual data. The question is now if using only two mask bits is enough to make the whole construction robust.

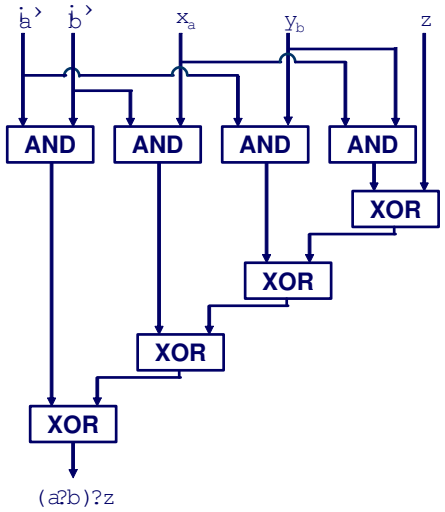

FIGURE 4. AND operation on masked data.

From 4.4 it follows that if we use only two existing mask bits, x_i and y_j to mask the value $(a_i \cdot b_j)$, then in order to obtain a robust mask, we would have to XOR the result of computations on masked data with the new mask $(x_i \oplus y_j)$, which can be achieved while computing the terms of the "mask correction".

Far better solution is to use a third random bit, z, as a new mask, computing a "masked" AND operation, for example, as follows

$$((\tilde{a} \cdot \tilde{b}) \oplus ((x_i \cdot \tilde{b}) \oplus ((x_i \cdot y_j) \oplus z))) \oplus (y_j \cdot \tilde{a}). \tag{4.5}$$

Alternative constructions can be used as well stemming from the basic algebraic formulas relating the operations AND, OR and XOR in $GF(2)$.

An actual implementation of the masked AND operation is obtained as a cascade of layers of logic gates, as shown in Figure 4.5. The first layer generates the

necessary elementary AND-terms. All other layers, including the last one, produce the variables in the form $(\alpha \oplus z)$, where z is independent on all the bits we have used so far.

5. Building Blocks for the Masked SubByte Transformation

From the details of the implementation of *SubByte* in the composite field $GF((2^4)^2)$ as a fully combinational logic design given in [29], we can see that the complexity of the S-box it 123 XOR gates in total, plus 16 multiplexors and a number of AND gates, which the designers ignored in their calculations.

However, for computations on masked data, AND operations are the most resource-consuming, as we had seen in the previous section. Hence, we have to take them into account.

When manipulating masked data, all stages in the *SubByte* transformation, apart from inversion in $GF(2^4)$, require a simple mask correction (either an analogous computations on masks performed in parallel, which duplicates the gates, or pipelined computations on masked data and masks (for which we would need 128-bits of additional registers. Hence, for a fully parallel implementation of all operations with the exception of the inversion in $GF(2^4)$, we need 222 XOR gates.

To implement a byte inversion, we have to compute inversion on a two-term polynomial (see equation 3.5), which takes 3 multiplication and one inversion in $GF(2^4)$.

From the formulas 3.6 it follows that an unmasked inversion in $GF(2^4)$ requires 12 XOR and 9 AND gates. For masked data, for each AND operation we need additionally 3 AND and 4 XOR gates, which makes in total 36 AND and 48 XOR gates to invert one element of $GF(2^4)$.

Multiplication in $GF(2^4)$ requires 16 AND and 12 XOR operations, and there are 3 multiplication units for each byte, which gives altogether 192 binary AND and 192+36 binary XOR gates.

Since there are 20 *SubByte* operations in one round, each requires inversion in $GF(2^8)$, the total number of gates depend on the design trade-offs. In a "minimalist" design with 1 S-box only, 653 XOR and about the same amount of AND gates is required. In a fully parallel design, one would need an equivalent of 13,000 XOR gates altogether to implement an inversion on masked values.

As we recall from [19], a gate equivalent required for an implementation of one *SubByte* transformation as a lookup table is 1,700. Considering that fully parallel implementation of one round on masked data would need either on-a-fly generation of 16 256-byte Sbox tables, or storing all 16 pre-computed modified Sbox tables, our solution is much better in terms of the gate count (about 25%).

6. Implementation of Inversion on Masked Data as a Table Lookup

A well-known software solution [17] consists in "masking" an original look-up table T which implements the *SubByte* transformation, with two masks, the input mask R^{in} and the output mask R^{out}, in such a way that for the modified table T^M the following holds:

$$T^M[A_{i,j} \oplus R^{in}_{i,j}] = T[A_{i,j}] \oplus R^{out}_{i,j}. \tag{6.1}$$

This implies that the masked table must be computed for each pair R_{in}, R_{out}. If one *fixes* the pair prior to each run of the AES computations and pre-computes table

lookups, then up to 20 256-byte tables should be stored in RAM, which requires $6KB$ of memory. If the new tables are computed "on-a-fly", for example, using a fast algorithm suggested in [28], as a sequence of table splits into blocks of certain size and block permutations depending on the mask value, the transformation still takes some time because it has to be done 16 (20, including the *Key Scheduling* operation) times per round!

Here is the "split-and-shift" algorithm for fast table re-computations.
Look-up table re-computation.

```
Input:  inverse masked table T + R_out;
        random M = (m_7, ..., m_1, m_0)
Output: table T' such that T'[b+M] = T[b] + r_b for b = 0..255,
   T' := T;
   For every m_i from (m_7, ..., m_0) in random order do:
      If m_i = 1 then
         (1) split T' into blocks, each block containing 2^(i)
             subsequent elements from T;
         (2) swap pairwise j-th and j+1-st blocks;
         (3) assign the result to T';
   Return T'
```

However, for an inversion in composite field $GF((2^4)^2)$ the idea of masking the lookup table seems attractive.

Indeed, from the architecture of the S-box operation in a composite field (see Fig. 3) it follows that the actual inversion is applied to the masked elements in $GF(2^4)$. There are only 16 possible values for a 4-bit mask. For each such value, the "mask correction" for the sequence of all linear operations that occur prior to the inversion, can be pre-computed once and for all. Hence, we would need a cross-reference table M, which for each possible value of the mask indicates the "correct" entry in the (randomized) inversion table.

In other words, $M[x_i] = I$, such that the following property holds: $T[I] = a_i^{-1}$, where T is the "original" inverse table in $GF(2^4)$. Of course, one immediately notice that such straightforward implementation would be vulnerable to DPA attack. Hence, instead of keeping the inversion table T, we have to keep 16 16×4-bit tables obtained from T according to the "split-and -shift" algorithm. Assuming that the output mask, z is fixed prior to running AES encryption/decryption, entries of each of the 16 tables kept in RAM are already masked.

Hence, the inversion is a two-step process. Given an input value $Aa_i \oplus x_i$ and a mask correction value x_i, we do the following

(1) Given an input x_i, lookup in the correspondence table M provides us with the index of one of the 16 "split-and-shift" inversion tables, i.e., $M[x_i] = j$, such that $T_j[a_i \oplus x_i] = a_i^{-1} \oplus z$.
(2) Given an index j and an input data $(a_i \oplus x_i)$, return the value $T_j[a_i \oplus x_i]$ as a result of a table T_j lookup.

Altogether, we need $16 \times 16 \times 4$ bits, or 128 bytes, for all 16 tables, plus additional 8 bytes for M. How many copies are used altogether depends on the design trade-offs: e.g., an architecture with 4 S-boxes requires 544 bytes of RAM.

7. Conclusion

In this paper we propose a new solution to the problem of hardware implementation of AES secure against DPA attacks. Namely, we designed a combinational logic block to compute inversion on masked data, without ever revealing the actual data bits in a process. Our solution is, in fact, rather general, and can be applied to other cryptographic algorithms. It is quite different from the dual rail logic [27] design, but provides comparable protection. Taking into account that dual rail logic is very hard to implement in real life, our design offers an alternative solution to hardware protection.

At this stage, we would refrain from claiming that we found the best solution to the problem of designing an efficient and secure AES co-processor for smart cards. Many other considerations, such as power consumption, throughput, production cost, etc., have to be taken into account.

As a future work, we would like to compare our novel hardware that works with masked data with "traditional" ASIC implementation of AES, where resistance to DPA attacks is ensured by well-known and widely used hardware counter measures, such as metal shields, random clock, and random current generator and their combinations. It is generally already known, for example, that clock randomization alone is not sufficient protection against DPA attacks because its effect can be relatively easy eliminated by clever processing of power curves using signal processing techniques. Similar reasoning can be applied to random current generators. However, it is not the "best" feature that counts, but overall design trade-offs. Investigation of such trade-offs is a challenging experimental problem.

References

[1] Akkar, M., Giraud, C.: An implementation of DES and AES, secure against some attacks. Proc. *Cryptographic Hardware and Embedded Systems: CHES 2001*, Volume 2162 of Lecture Notes in Computer Science, pp. 309-318, Springer-Verlag, 2001.

[2] R. Anderson, M. Kuhn, *Low cost attacks on tamper resistant devices*, In M. Loman et al. (eds.), Security Protocols, proceedings 5th International Workshop IWSP, volume 1361 of Lecture Notes in Computer Science, pp. 125-136, Springer-Verlag, 1997.

[3] M. Bucci, L.Germani, M. Guglielmo, R. LUzzi, A. Trifiletti, *A simulation methodology for DPA resistance testing of cryptographic processors*, manuscript, 2003.

[4] D. Boneh, R. DeMillo, R. Lipton, *On the importance of checking cryptographic protocols for faults*, In proceedings of Advances in Cryptology – Eurocrypt'97, volume 1233 of Lecture Notes in Computer Science, pp. 37-51, Springer-Verlag, 1997.

[5] E. Biham, A. Shamir, *Differential fault analysis of secret key cryptosystems*, In volume 1294 of Lecture Notes in Computer Science, pp. 513-525, Springer-Verlag, 1997.

[6] Chari, S., Jutla, C., Rao, J., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. Proc. *Advances in Cryptology – Crypto'99*, volume 1666 of Lecture Notes in Computer Science, pp. 398-412, Springer-Verlag, 1999.

[7] Daemen, J., Rijmen, V.: *The design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag Berlin Heidelberg, 2002

[8] S. Fruhauf, L. Sourge, *Safety device against the unauthorized detection of protected data*, in 1990 [8]. U.S. patent number 5,404,402; April4, 1995.

[9] Gandolfi, K., Mourtel, C., Oliver, F.: Electromagnetic analysis: concrete results. Proc. *Cryptographic Hardware and Embedded Systems: CHES 2001*, Volume 2162 of Lecture Notes in Computer Science, pp. 251-261, Springer-Verlag, 2001.

[10] Goubin, L., Patarin, J.: DES and differential power analysis. Proc. *Cryptographic Hardware and Embedded Systems: CHES'99*, volume 1717 of Lecture Notes in Computer Science, pp. 158-172, Springer-Verlag, 1999.

[11] P. Kocher, J. Jaffe, B. Jun, *Differential power analysis*, In Advances in Cryptology – CRYPTO'99, volume 1666 of Lecture Notes in Computer Science, pp. 388-397, Springer-Verlag, 1999.

[12] Kocher, P.: Timing attacks on implementations of Diffie-Hellmann, RSA, DSS, and other systems. In Proc. *Advances in Cryptology – Crypto'96*, volume 1109 of Lecture Notes in Computer Science, pp. 104-113, Springer-Verlag, 1996.

[13] P. Kocher, J. Jaffe and B. Jun, *Using unpredictable information to minimize leakage from smartcards and other cryptosystems*, USA patent, International Publication number WO 99/63696, December 9, 1999.

[14] O. Kommerling, M. Kuhn, *Design principles for tamper-resistant smartcard processors*, In proceedings USENIX Workshop on Smartcard Technology (Smartcard 99), pp. 9-20, 1999.

[15] C. C. Lu, S-Y. Tseng, *Integrated design of AES (Advanced Encryption Srandard) encryptor and decryptor*, In proceedings IEEE conf. on Application-Specific Systems, Architectures, and Processors (ASAP'02), pp. xxx, IEEE, 2002.

[16] S. Mangard, M. Aigner, S. Dominikus, *A highly regular and scalable AES hardware architecture*, IEEE Transactions on Computers, vol. 52, no. 4, pp. 483-491, 2003.

[17] Messerges, T.: Securing the AES finalists against power analysis attacks. Proc. *Fast Software Encryption Workshop 2000*, volume 1978 of Lecture Notes in Computer Science, pp. 150-165, Springer-Verlag, 2000.

[18] T. S. Messerges, E. A. Dabbish, R. H. Sloan, *Examining smart-card security under the thread of power analysis*, IEEE Trans. Computers, vol. 51, no. 5, pp. 541-522, May 2002.

[19] S. Morioka, A. Satoh, *An optimized S-Box circuit architecture for low power AES design*, 2002.

[20] C. Paar, *Efficient VLSI architectures for computations in Galois fields*, PhD Thesis, Linkoping University, Linkoping, Sweden, 1991.

[21] J. J. Quisquater, D. Samide, *Side channel cryptanalysis*, In proceedings of the SECI 2002, pp. 179-184, 2002.

[22] J. J. Quisquater, D. Samide, *Electromagnetic analysis (ema): measures and countermeasures for smart cards*, In proceedings of Smarccard Programming and Secvurity, volume 2140 of Lecture Notes in Computer Science, pp. 200-210, Springer-Verlag, 2001.

[23] V. Rijmen, *Efficient implementation of Rijndael SBox*, http://www.esat.kuleuven.ac.be/ rijmen/rijndael

[24] A. Rudra, P. Dubey, C. Julta, V. Kumar, J. Rao, P. Rohatgi, *Efficient Rijndael implementation with composite field arithmetic*, In Cryptographic Hardware and Embedded Systems – CHES'01. volume 2162 of Lecture Notes in Computer Science, pp. 175-188, Springer-Verlag, 2001.

[25] A. Satoh, S. Morioka, K. Takano, S. Munetoh, *A compact Rijndael hardware architecture with S-Box optimization*, In Advances in Cryptology – ASIACRYPT 2001. Volume 2248 of Lecture Notes in Computer Science, pp. 239-254, Springer-Verlag, 2001.

[26] E. Sprunk, *Clock frequency modulation for secure microprocessors*, USA patent number WO 99/63696, December 9, 1999.

[27] K. Tiri, M.Akmal, I. Verbauwhede, *A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards*, In proceedings of IEEE 28th Europen Solid-State Circuit Conf. – ESSCIRC'02, 2002.

[28] E. Trichina, D. de Seta, L. Germani *Simplified Adaptive Multiplicative Masking for AES and its secure implementation*, *Cryptographic Hardware and Embedded Systems: CHES 2002*, Volume 2523 of Lecture Notes in Computer Science, pp. xxx, Springer-Verlag, 2002.

[29] J. Wolkerstorfer, E. Oswald, M. Lamberger, *An ASIC implementation of the AEs S-Boxes*, In Topic in Cryptography – CT-RSA. Volume 2271 of Lecture Notes in Computer Science, pp. 67-78, Springer-Verlag, 2002.

Smart Card System Engineering, System LSI Division, Device Solutions Network, Samsung Electronics Co. LTD

E-mail address: e.v.trichina@samsung.com