An Efficient Masking Scheme for AES Software Implementations*

Elisabeth Oswald[1] and Kai Schramm[2]

[1] Institute for Applied Information Processing and Communciations (IAIK), TU Graz, Inffeldgasse 16a, A–8010 Graz, Austria
[2] Horst Görtz Institute for IT Security (HGI), Universitätsstr. 150, Ruhr University Bochum, Germany, 44780 Bochum, Germany
elisabeth.oswald@iaik.tugraz.at,
schramm@crypto.ruhr-uni-bochum.de

Abstract. The development of masking schemes to secure AES implementations against power-analysis attacks is a topic of ongoing research. The most challenging part in masking an AES implementation is the SubBytes operation because it is a non-linear operation. The current solutions are expensive to implement especially on small 8-bit processors; they either need many large tables or require a large amount of operations. In this article, we present a masking scheme that requires considerably less tables and considerably less operations than the previously presented schemes. We give a theoretical proof of security for our scheme and confirm it with actually performed DPA attacks.

1 Introduction

The *Advanced Encryption Standard* (short: AES) [Nat01] is the worldwide defacto standard for symmetric encryption. It succeeds the older *Data Encryption Standard* (short: DES) [Nat99]. Therefore, it will be used in manifold services ranging from high-performance applications such as web services to low-cost (low memory, low power consumption) implementations on smart cards. Especially in the case of software implementations for smart cards limited memory (ROM, RAM, XRAM) poses a challenging constraint for implementors. Even worse, implementation attacks such as differential power analysis attacks (short: DPA attacks) [KJJ99] require considerable effort from the implementor's side to come up with implementations that do not succumb to such attacks.

During the past years, a lot of effort has been devoted to the research in DPA attacks. It has become clear that smart cards without built in countermeasures are highly susceptible to all kinds of DPA attacks. Hence, researchers have proposed all kinds of schemes to secure implementations of different kinds of cryptographic algorithms. The AES algorithm has received the largest attention amongst symmetric schemes because of its expected widespread use.

* The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

J. Song, T. Kwon, and M. Yung (Eds.): WISA 2005, LNCS 3786, pp. 292–305, 2006.

In this article, we focus on the scenario where AES is implemented in software on 8-bit platforms such as commonly available smart cards. We propose a masking scheme for this scenario which requires less tables, *i.e.*, less memory, and less operations than comparable schemes in the same scenario.

The remainder of this article is organized as follows. In Sect. 2, we give a brief overview of AES. In Sect. 3, we review the problem of masked AES implementations on restricted platforms and we survey related work. In Sect. 4, we introduce our new scheme and provide a theoretical analysis of its security against DPA attacks. In Sect. 5, we describe our implementation of our new scheme on an 8-bit smart card. In Sect. 6, we report on the results of practical DPA attacks that we have performed on our implementation. We conclude this article in Sect. 7.

2 Advanced Encryption Standard

The AES is a symmetric cipher which encrypts/decrypts data with a block size of 128 bits using a key of size 128, 192 or 256 bits. In the following we will briefly decribe the encryption scheme of AES. The decryption scheme is equivalent but uses the inverse transformations. The 16-byte plaintext $p_0 p_1 ... p_{15}$ is arranged in four-by-four byte matrix, called *state*. All transformations in AES operate on the state.

p_0	p_4	p_8	p_{12}
p_1	p_5	p_9	p_{13}
p_2	p_6	p_{10}	p_{14}
p_3	p_7	p_{11}	p_{15}

The following transformations are used in the AES cipher:

1. **AddRoundKey:** A round key is added to the state matrix using the XOR operation. The round keys are derived from the key with the *Key Expansion* algorithm.
2. **ShiftRows:** The second row of the state matrix is cyclically shifted by one byte to the left, the third row by two bytes and the fourth row by three bytes. The first row remains unchanged. The ShiftRows transformation increases the *diffusion* properties of AES.
3. **SubBytes:** Each byte of the state matrix is substituted using a bijective substitution box (short: S-box). The S-box is based on the non-linear inversion in the finite field $GF(2^8)$ and a bitwise affine transformation. The S-box step increases the *confusion* properties of AES.
4. **MixColumns:** The MixColumns step is a linear transformation, which increases the *diffusion* properties of AES. Each column is mixed using the following matrix multiplication:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

where b_i are the bytes of the input column, c_i are the bytes of the output column, and matrix elements {03}, {02} and {01} correspond to the polynomials $x + 1$, x and 1.

5. **Key Expansion:** The key expansion derives the round keys from the cipher key.

The AES encryption scheme is given below:

```
AddRoundKey

for round = 1 to Nr
    SubBytes
    ShiftRows
    MixColumns
    AddRoundKey
end

SubBytes
ShiftRows
AddRoundKey
```

The Key Expansion is typically performed interleaved with the rounds in software. The number of rounds `Nr` depends on the key size. If the key size is 128, 192 or 256 bits 10, 12 or 14 rounds are used, respectively. All AES transformations but SubBytes are linear. Hence, only SubBytes requires special attention with regard to masking. This articles intensely focuses on a secure yet efficient software implementation of SubBytes.

3 Related Work

In a typical software implementation, the SubBytes operation is implemented as a table look-up. Hence, for an input value in of a SubBytes operation, the output is derived as $out = S(in)$. As there are 16 8-bit chunks in the AES state, 16 table look-up operations have to be performed in one encryption round (not taking the key schedule into account).

When we mask the SubBytes operation with a value m (the mask), *i.e.*, when we add a random value m' (the mask) to its input, we have to re-compute the table S such that $out = S(in) + m$, where in is masked,*i.e.*, $in = x + m'$. Hence, we need a table $MSubBytes()$ such that

$$\mathbf{MSubBytes}(\mathbf{x} + \mathbf{m}') = \mathbf{SubBytes}(\mathbf{x}) + \mathbf{m}.$$

The **MSubBytes()** table for the masks m and m' (for simplicity, m is often chosen to be equal to m') is calculated according to Algorithm 1. The exclusive-or (short: XOR) operation is denoted by + in this article.

When more than one mask m is to be used, more $MSubBytes()$ tables need to be computed. For example, when using 16 masks m, 16 tables are needed.

Algorithm 1. Computation of Masked **SubBytes**

Require: m, m'
Ensure: $\mathbf{MaskedSubBytes}(\mathbf{x} + \mathbf{m}') = \mathbf{SubBytes}(\mathbf{x}) + \mathbf{m}$,
1: **for** $i = 0$ to 255 **do**
2: $MaskedSubBytes(i + m') = \mathbf{Subbytes}(\mathbf{i}) + m$
3: **end for**
4: Return($MaskedSubBytes$)

As stated in [GT03], the usage of the same mask for all 16 s-boxes represents a serious threat, because intermediate variables (e.g. the s-box outputs) are masked with the same mask and their mutual correlation can be used to apply second order DPA attacks.

For this article, we consider the scenario where AES is implemented on an 8-bit smart card. We assume that AES is not used for bulk encryption. Instead, AES is used for example in a challenge-response protocol, where only one instance of the algorithm is typically computed at a time.

For every mask m, a masked table needs to be computed. There are several strategies an implementor can follow. Either all 256 masked tables are precomputed and stored in a memory, or, only t tables for the t 8-bit masks are pre-computed at the beginning of the AES algorithm and stored in memory. Another option is to compute the masked table on the fly whenever it is needed during the encryption algorithm.

We argue that in practice the second method is the most attractive one, because it gives the best tradeoff between the amount of memory and the number of operations. Remember that the size of one table is 256 bytes. Counting the number of operations for this algorithm for t masks shows that in total an amount of $2 \times t \times 256$ table look-ups read/write operations and $2 \times t \times 256$ XOR operations are needed. In total, $(t + 1) \times 256$ bytes of memory are used. In typical AES implementations, a separate mask for each 8-bit chunk would be used. That amounts then to 8192 table look-ups, 4352 bytes of memory and 8192 XOR operations.

Many algorithmic countermeasures have been proposed for the AES algorithm, see [AG01], [GT03], [TSG03], [TK04], [BGK05] and [OMPR05]. They are all based on masking the intermediate value, *i.e.*, adding a random number (the mask) to the intermediate AES values. However two of them, [AG01] and [TSG03], are both susceptible to a certain type of (first-order) differential side-channel attack, the zero-value attack. The latter one has turned out to be vulnerable even to standard differential side-channel attacks [ABG04].

The countermeasure presented in [GT03] leads to very costly implementations. This is due to the fact that in order to circumvent the zero-value problem, the authors propose to embed the inversion operation (which is part of SubBytes) into a larger algebraic structure such that the zero-value is mapped to different non-zero values. Although this construction is mathematically elegant, implementations thereof, especially on 8-bit platforms, are not.

The countermeasure presented in [TK04] uses pre-computed discrete logarithm and exponentiation tables to realize the SubBytes operation (*i.e.*, the

inversion operation that is part of the mathematical description of SubBytes). This approach is based on the fact that a non-zero element in a finite field can be inverted by computing the logarithm of the element to a particular base[1] and exponentiating the base again with the negated logarithm. The inversion of the zero element has to be carefully taken into account by using a conditional check, e.g. the authors suggest to manipulate the discrete logarithm and exponentiation tables in such a way that the zero element is inverted correctly to itself. Unfortunately, we believe that this approach has a flaw which is linked to the inversion of the zero element. In their work, the authors state that conditional branching for the zero element can be avoided by changing two table elements: $log[0] = 2^n - 1$ and $alog[2^n - 1] = 0$. However, because an inversion is defined as

$$\alpha^{-1} = alog[(2^n - 1) - log[\alpha]]$$

the inversion of zero will result in $0^{-1} = alog[0] = 1 \neq 0$ and, moreover, the inversion of 1 will result in $1^{-1} = alog[(2^n - 1) - log[1]] = alog[2^n - 1] = 0 \neq 1$. As a matter of fact, by setting $log[0] = 0$, $log[1] = 2^n - 1$ and $alog[2^n - 1] = 0$, we found a possibility to correct the *log* and *alog* tables in such a way that both inversions will work properly, again. In their paper, a multiplication of two elements is defined as

$$\alpha \cdot \beta = alog[log[\alpha] + log[\beta] \ \texttt{mod}\ 2^n - 1]$$

However, when using this method a multiplication with zero will only always result in zero, if conditional branching is used. Based on the inversion and multiplication with the *log* and *alog* tables the authors propose two different masking schemes which are supposed to provide a secure inversion. We have carefully implemented and tested both schemes. We observed that in both schemes there occur special cases when the s-box input, the mask or masked, intermediate variables are equal to zero and which will result in a faulty behavior of the proposed masking schemes. We believe that a correction of their approach is only possible with the use of conditional branches, which makes it susceptible to power-analysis attacks.

The countermeasures presented in [BGK05] and [OMPR05] are based on a similar idea. In both papers, the authors assume that the inversion operation is computed step-by-step, either as exponentiation or with composite field arithmetic. The exponentiation method is advertised for software implementations and described in [BGK05]. The composite-field method is advertised for hardware implementations and is described in detail in [OMPR05]. Both methods do not seem to be particularly suited for 8-bit software implementations. However, as we will show in this article, especially the composite-field method can be adapted in such a way that it is suitable for 8-bit platforms.

4 A New Scheme for Efficiently Masking AES in Software

The only difficult part in masking AES is to mask the SubBytes operation. The SubBytes operation is composed of two parts: an inversion in $GF(2^8)$ and an

[1] i.e. for a chosen generator.

affine mapping. Again, masking the affine part is easy, so we focus on the non-linear inversion operation only. Our goal is that all input and output values in the computation of the inverse are masked. According to [OMPR05], a masked input can be transformed to the composite field $GF(2^4) \times GF(2^4)$ with an isomorphic mapping, where it can be securely and efficiently inverted, and finally transformed back to the $GF(2^8)$. The inversion operation in the composite field can be computed as follows:

$$((a_h+m_h)x+(a_l+m_l))^{-1} = (a'_h + m'_h)x + (a'_l + m'_l) \tag{1}$$

$$\begin{aligned} a'_h + m'_h &= f_{a_h}((a_h + m_h), (d' + m'_d), m_h, m'_h, m'_d) \\ &= a_h \times d' + m'_h \end{aligned} \tag{2}$$

$$\begin{aligned} a'_l + m'_l &= f_{a_l}((a'_h+m'_h), (a_l+m_l), (d'+m'_d), m_l, m'_h, m'_l, m'_d) \\ &= (a_h + a_l) \times d' + m'_l \end{aligned} \tag{3}$$

$$\begin{aligned} d + m_d &= f_d((a_h + m_h), (a_l + m_l), p_0, m_h, m_l, m_d) \\ &= a_h^2 \times p_0 + a_h \times a_l + a_l^2 + m_d \end{aligned} \tag{4}$$

$$\begin{aligned} d' + m'_d &= f_{d'}(d + m_d, m_d, m'_d) \\ &= d^{-1} + m'_d \end{aligned} \tag{5}$$

The functions f_{a_h}, f_{a_l}, f_d and $f_{d'}$ are functions on $GF(2^4)$.

This calculation of a masked inversion operation is based on the composite field approach that is described in detail in [WOL02].

Whereas in [OMPR05] this approach is applied to hardware implementations and has been extended to work in so-called tower fields, we pursue a different approach. We show that these formulae can be mapped to a sequence of table look-ups and XOR operations. We show how to define tables which only require little space in memory. Furthermore, we show that only a small number of table look-ups are required to calculate the formulae.

4.1 Pre-computed Tables

We compute a number of tables that do the operations in $GF(2^4)$ and store them in memory:

$$\begin{aligned} T_{d_1} &: ((x + m), m) \mapsto x^2 \times p_0 + m \\ T_{d_2} &: ((x + m), (y + m')) \mapsto ((x + m) + (y + m')) \times (y + m') \\ T_m &: ((x + m), (y + m')) \mapsto (x + m) \times (y + m') \\ T_{inv} &: ((x + m), m) \mapsto x^{-1} + m. \end{aligned}$$

All tables (or functions) take two elements of $GF(2^4)$ as inputs and give an element of $GF(2^4)$ as output.

With those 4 Tables, we can compute formulas (2)-(5). In order to map $GF(2^8)$ elements to $GF(2^4) \times GF(2^4)$ elements and vice versa, we need two

more tables $Map : x \mapsto z$ and $Map^{-1} : z \mapsto x$. Map takes an element x of $GF(2^8)$ as input and gives an element z of $GF(2^4) \times GF(2^4)$ as output. Map^{-1} works vice versa. We assume that for all tables the input masks and the output masks are identical. Hence, the size of one table is at most 256 bytes and so we can pre-compute all tables and store them in read-only memory (ROM), since there is no need to compute them during run-time. This is a significant advantage over the use of $MSubBytes()$ tables. They have to be computed for every new mask m during run-time or at least at the invocation of a new AES encryption run.

4.2 Masked Inversion

First, we have to compute the masked value of d, *i.e.*, $d+m_d = d+m_h$ according to (4):

$$\begin{aligned} f_d(a_h + m_h, a_l + m_l, m_h, m_l, m_h) = {} & T_{d_1}(a_h + m_h, m_h) \\ & + T_{d_2}((a_h + m_h), (a_l + m_l)) \\ & + T_m((a_h + m_h), m_l) \\ & + T_m((a_l + m_l), m_h) + T_m((m_h + m_l), m_l). \end{aligned} \tag{6}$$

It is easy to check that the result will be indeed $a_h^2 \times p_0 + a_h \times a_l + a_l^2 + m_h$. For this computation we need five table look-up operations (TLs), four XOR operations and an additional XOR operation to compute $(m_h + m_l)$ which is used as input in $T_m((m_h + m_l), m_l)$. Note that the results of $T_m((a_h + m_h), m_l)$ and $T_m((a_l + m_l), m_h)$ are used again in (8) and (9), respectively, therefore it is a good idea to store these results and reuse them later on in order to save these two look-up operations.

In the next step we compute the inverse of the masked d with one more table look-up operation:

$$f_{d'}(d + m_h, m_h, m_h) = T_{inv}(d + m_h, m_h). \tag{7}$$

In order to derive $f_{a_h}()$, we first compute $d^{-1} + m_l$ by one XOR addition with the term $(m_h + m_l)$. Then $f_{a_h}(a_h + m_h, d^{-1} + m_l, m_h, m_h, m_l)$ can be computed as follows:

$$\begin{aligned} f_{a_h}(a_h + m_h, d^{-1} + m_l, m_h, m_h, m_l) = {} & T_m(a_h + m_h, d^{-1} + m_l) \\ & + m_h + T_m(d^{-1} + m_l, m_h) \\ & + T_m(a_h + m_h, m_l) + T_m(m_h, m_l). \end{aligned} \tag{8}$$

This computation gives as output $a_h \times d^{-1} + m_h$. For this computation, we need three new table look-up operations and four XOR operations in total.

In the last step we derive $f_{a_l}(a_h \times d^{-1} + m_h, a_l + m_l, d^{-1} + m_l, m_l, m_h, m_l, m_l)$. Hence, we calculate:

$$
\begin{aligned}
&f_{a_l}(a_h \times d^{-1} + m_h, a_l + m_l, d^{-1} + m_l, m_l, m_h, m_l, m_l) \\
&= T_m((a_l + m_l), (d^{-1} + m_h)) + m_l + T_m(d^{-1} + m_h, m_l) \\
&+ T_m(a_l + m_l, m_h) + f_{a_h} + m_h + T_m(m_h, m_l). \qquad (9)
\end{aligned}
$$

This gives $a_l \times d^{-1} + a_h \times d^{-1} + m_l$ as a result. Note that the term $T_m(m_h, m_l)$ occurs in the computation of f_{a_h} and f_{a_l}.

Hence, by also storing $f_{a_h} + T_m(m_h, m_l)$ during the computation of f_{a_h} and using this term during the computation of f_{a_l}, one additional table look-up and one XOR can be saved. Therefore, for this computation we only need two additional table look-ups and five XOR operations.

Prior to the inversion in $GF(2^4) \times GF(2^4)$ we need to map the 8-bit values (elements in $GF(2^8)$) to 2×4-bit values (elements in $GF(2^4) \times GF(2^4)$). This is done by a table look-up as well. Mapping back from $GF(2^4) \times GF(2^4)$ to $GF(2^8)$ can be achieved with an additional look-up table. Moreover, it makes sense to combine the isomorphic mapping from $GF(2^4) \times GF(2^4)$ to $GF(2^8)$ with the affine transformation that is part of SubBytes and use only one table for both.

Total Costs of a Masked Inversion. If we review the number of table look-ups (TLs) and XOR additions required for an entire masked AES SubBytes operation, we need five TL operations and four XOR additions in (6), one TL operation in (7), three TL operations and four XOR additions in (8), two TL operations and five XOR additions in (9). Furthermore, we need three TL operations for the isomorphic transformations: two TL operations to map the masked inversion input and the mask to $GF(2^4) \times GF(2^4)$ and one TL operation to map the masked result of the inversion back to $GF(2^8)$ and perform the affine transform.

This sums up to a total of 14 table look-up operations and 15 XOR operations.

4.3 Theoretical Security Analysis

In this section we show that all data-dependent intermediate masked values that are computed during the masked inversion operation are statistically independent from the unmasked values.

Hence, we follow the definition of security that was introduced in [CJRR99] and strengthened in [BGK05].

The values that we have to investigate are the outputs of the functions (tables) T_{d_i}, T_m, T_{inv}, Map, Map^{-1} and all intermediate values that occur after an XOR operation. In [OMPR05] it has been shown in Lemma 5 that a sum of independent masked values will again be independent from the unmasked values as long as an independent mask is used during the summation. Furthermore, in Lemmas 1–4 it has been shown that the XOR operation, as well as the masked multiplication and the masked squaring are secure in the sense that their output is statistically independent from the plaintext input.

Lemma 1. *Let $x \in GF(2^n)$ be arbitrary and let $p_0 \in GF(2^n)$ be an arbitrary but fixed value. Let $m \in GF(2^n)$ be independently and uniformly distributed in $GF(2^n)$. Then $T_{d_1}(x + m, m) = x^2 \times p_0 + m$ is uniformly distributed regardless of x. Therefore, the distribution of $x^2 \times p_0 + m$ is independent of x.*

Proof. As x is an element of the binary extension field, the element $x^2 = (\sum_i a_i \alpha^i)^2 = \sum_i a_i \alpha^{2i}$ with $a_i \in \{0,1\}$ is in $GF(2^n)$ as well. Hence, all elements of $GF(2^n)$ are quadratic residues and thus x^2 is uniformly distributed on $GF(2^n)$. Consequently, also $x^2 \times p_0$ and $x^2 \times p_0 + m$ are uniformly distributed.

For the independency of the output of T_{d_2} we reuse Lemma 2 of [BGK05].

Lemma 2. *Let $x, y \in GF(2^n)$ be arbitrary. Let $m, m' \in GF(2^n)$ be independently and uniformly distributed in $GF(2^n)$. Then the probability distribution of $T_m(x+m, y+m') = (x+m) \times (y+m')$ is*

$$\Pr((x+m) \times (y+m') = i) = \begin{cases} \frac{2^{n+1}-1}{2^{2n}} & , \text{if } i = 0, \text{ i.e., if } m = x \text{ or } m' = y \\ \frac{2^n-1}{2^{2n}} & , \text{if } i \neq 0. \end{cases}$$

Therefore, the distribution of $(x+m) \times (y+m')$ is independent of x and y.

Lemma 3 follows directly from Lemma 2 and the observation that all elements of $GF(2^n)$ are quadratic residues.

Lemma 3. *Let $x, y \in GF(2^n)$ be arbitrary. Let $m, m' \in GF(2^n)$ be independently and uniformly distributed in $GF(2^n)$. Then the probability distribution of $T_{d_2}(x+m, y+m') = (x+m) \times (y+m') + (y+m')^2$ is*

$$\Pr((x+m) \times (y+m') + (y+m')^2 = i) = \begin{cases} \frac{2^{n+1}-1}{2^{2n}}, & \text{if } i = 0, \text{ i.e., if } m = x \text{ or } m' = y \\ \frac{2^n-1}{2^{2n}}, & \text{if } i \neq 0. \end{cases}$$

Therefore, the distribution of $(x+m) \times (y+m')$ is independent of x and y.

The independence of $T_{inv}(x+m, m) = x^{-1} + m$ is clear as the inversion operation is bijective (note that the zero element is mapped to the zero element) and the XOR of any $a + m$ is independent from a. The mappings between $GF(2^8)$ and $GF(2^4) \times GF(2^4)$ are bijections and therefore their masked output is independent from the unmasked input in a statistical sense.

Based on these results we may conclude that the algorithm for computing masked inversion complies to the definition of security used in [BGK05].

Recently it was discovered, see [MPG05] and [SSI04], that glitches in CMOS circuits make masked implementations vulnerable to standard DPA attacks. Our masking scheme is also secure when glitches occur in a circuit as we only use table look-ups and XOR operations. For both operations it has been shown that glitches do not have an effect on their security, see [MPG05].

5 Implementation of the New Scheme

In our following analysis we regard the implementation of the SubBytes transformation in assembly on a smart card based on the 8-bit RISC architecture. In total, we require six pre-computed tables which can be stored in read-only memory (ROM). Table T_{d_1} takes two $GF(16)$ elements as input and gives one

$GF(16)$ element as output. The same holds for T_{d_2}, T_m and T_{inv}, as well. Hence, these four tables map an 8-bit input to a 4-bit output value.

In a practical software implementation there are two possibilities how the tables T_{d_1}, T_{d_2}, T_m and T_{inv} can be stored in memory on an 8-bit architecture. In a compact representation, each byte of these four tables stores two 4-bit output values, hence, each table requires 128 bytes in ROM and the four tables altogether require $4 \times 128 = 512$ bytes in ROM. The disadvantage of this compact representation is based on the fact that a few instructions are required after each table look-up to either erase the unwanted upper 4-bit half or to shift the upper 4-bit half by four bits to the right in order to erase the unwanted lower 4-bit half. These instructions are not required, if each byte of the tables T_{d_1}, T_{d_2}, T_m and T_{inv} only stores a single 4-bit result and the upper 4-bit half is always set to zero. This representation is more efficient in terms of clock cycles, but requires $4 \times 256 = 1024$ bytes in ROM. In the following we will only regard the efficient representation. The two isomorphic mappings from $GF(2^8)$ to $GF(2^4) \times GF(2^4)$ and back from $GF(2^4) \times GF(2^4)$ to $GF(2^8)$ deliver a $GF(2^4) \times GF(2^4)$ and a $GF(2^8)$ element as output, i.e. these two tables map an 8-bit input to an 8-bit output. Hence, in total we need $4 \times 256 + 2 \times 256 = 1536$ bytes to store all six tables in ROM.

The smart card architecture is a RISC design and provides 32 internal registers. A TL operation which reads an 8-bit value from a table stored in ROM to an internal register takes five clock cycles. A TL operation which reads an 8-bit value from a table stored in RAM to an internal register or writes an 8-bit value to a table stored in RAM takes four clock cycles. The XOR addition of two internal registers requires only a single clock cycle.

In an unmasked AES software implementation every SubBytes step would only require a single TL operation. If a standard masked table look-up, such as described in Sect. 3 is used, the SubBytes table would be stored in ROM and the masked tables would be derived from it prior to an AES encryption/decryption and then stored in RAM. If only one encryption is performed, this pre-computation would very likely be done for the 16 masks, only, and thus require $16 \times 256 = 4096$ bytes in RAM. During the encryption/decryption of AES only a single TL operation would be required for each SubBytes step. However, the pre-computation of the each masked table in RAM would require 256 XOR additions to mask the table index, 256 TL operations to read the unmasked table entries from ROM, 256 XOR additions to mask the table entries and finally 256 TL operations to store the masked table in RAM. If tables are generated in such a way for 16 different masks, this will result in pre-computational costs of $16 \times (256 + 256 \times 5 + 256 + 256 \times 4) = 45056$ clock cycles. If several encryption operations would be performed after each other and the same set of masks is used over and over again, the pre-computational costs occur only once. However, from a security point of view it is advisable to update the masks as often as possible. Another possibility is to store all masked tables in ROM. However, this would require $256 \times 256 = 64$ KB in ROM which might exceed the limitations in constrained environments such as smart cards.

Table 1. Comparison of various AES software implementations with regard to code size and speed for a single encryption

	ROM	RAM	PRE-TL	PRE-XOR	TL	XOR	cycles
unmasked	256	0	0	0	160	0	800
256 fixed masks	64 KB	0	0	0	160	0	800
single mask	256	256	512	512	160	0	3456
16 masks	256	4096	8192	8192	160	0	45696
MOS-box	1536	0	0	0	2240	2400	13600

As stated in Sect. 4, when using our proposal an entire SubBytes step for an arbitrary mask requires 14 TL operations and 15 XOR additions which results in $14 \times 5 + 15 = 85$ clock cycles. For an entire AES encryption this results in $10 \times 16 \times 85 = 13600$ clock cycles. Our method requires 1536 bytes in ROM and no RAM, moreover, no pre-computation needs to be performed. In Tab. 1 the costs of various masked and unmasked AES implementations are compared. Our proposal is referred to as "'New"' in Tab. 1.

Hence, the complexity of our proposal is lower in terms of memory and operations for a single encryption. If only a single mask is used, our proposal is about four times slower for a single encryption, however, our approach does not require any RAM. Furthermore, it has been pointed out in [GT03] that the usage of a single mask in AES may allow simple second-order DPA attacks, which can be avoided by the usage of 16 different masks in each round. If encryptions are repeated several times with the same set of 16 masks, our proposal will be slower after four encryptions, but will always require less memory.

6 Power Analysis of the New Scheme

In order to confirm the security claims that we made in Sect. 4 and to assess the practical security of our implementation, we performed DPA attacks on an AES

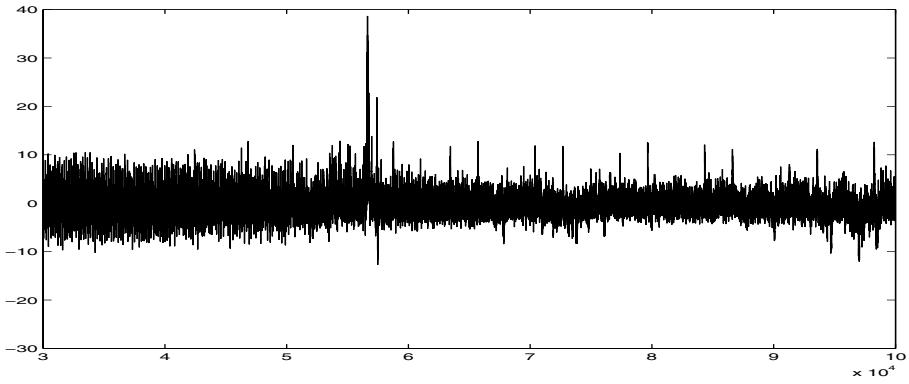

Fig. 1. DPA of the AES with no active countermeasure

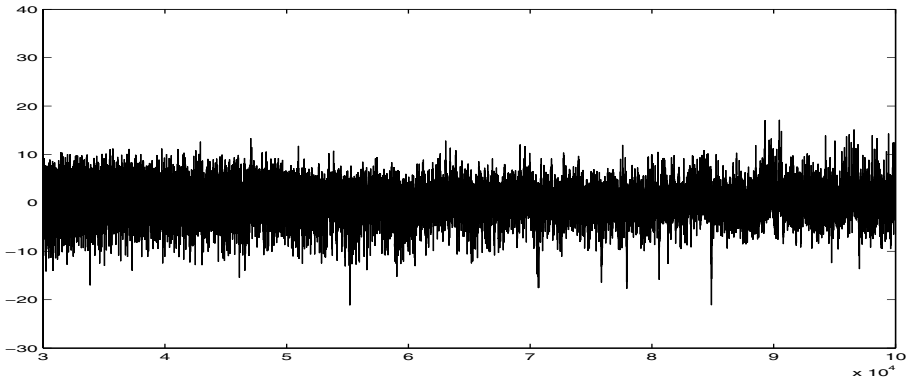

Fig. 2. DPA of the AES with our new masked s-box scheme

implementation based on our new inversion scheme. The target hardware was an 8-bit smart card. DPA attacks were performed in two independent experiments. The first time we performed DPA attacks on the implementation with the masking countermeasure switched off, i.e. all mask were fixed to zero. The second time we performed DPA attacks on the implementation with the masking countermeasure being active, i.e. all masks were randomly generated. In both experiments 1000 random plaintexts were encrypted and the corresponding power traces were measured using a digital oscilloscope with a sampling rate of 100 MSa/s and a current probe. The resulting differential traces are shown in Fig. 1 and Fig. 2.

It is obvious that the DPA of the unprotected AES implementation is successful, since a distinct correlation peak is contained in Fig. 1 for the correct key hypothesis. However, as shown in Fig. 2 the DPA of our new protected AES scheme was not successful.

7 Conclusion

In this article we have presented a new masking scheme for software implementations of AES on 8-bit platforms. Our scheme is based on computing the inverse operation, which is part of SubBytes, with composite-field arithmetic. All steps that are needed throughout the computation are done via table look-ups and XOR operations. We have proven that all intermediate masked values that occur during the computation are independent from unmasked intermediate values. We have confirmed our theoretical proof with actually performed DPA attacks. Our scheme is even secure when glitches in the underlying CMOS circuit occur because it only uses table look-ups and XOR operations. The strong point of our scheme is based on the fact that it is possible to use different masks for all 16 SubBytes operations with no RAM requirements. We believe that this is important, since RAM is generally very sparse on embedded devices such as smart

cards. Hence, our scheme provides a nice tradeoff between memory requirements and speed and seems to be well suited for small platforms.

Acknowledgements

We would like to thank Andreas Krügersen for implementing the new AES inversion scheme in assembly on the smart card.

References

[ABG04] Mehdi-Laurent Akkar, Régis Bevan, and Louis Goubin. Two Power Analysis Attacks against One-Mask Methods. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 332–347. Springer, 2004.

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.

[BGK05] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably Secure Masking of AES. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2005.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[GT03] Jovan D. Golić and Christophe Tymen. Multiplicative Masking and Power Analysis of AES. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2535 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

[Nat99] National Institute of Standards and Technology (NIST). FIPS-46-3: Data Encryption Standard, October 1999. Available online at `http://www.itl.nist.gov/fipspubs/`.

[Nat01] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.

[OMPR05] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A Side-Channel Analysis Resistant Description of the AES S-box. In Helena Handschuh and Henri Gilbert, editors, *Fast Software Encryption, 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Proceedings*, volume 3557 of *Lecture Notes in Computer Science*, pages 425–435. Springer, 2005. to appear.

[SSI04] Daisuke Suzuki, Minoru Saeki, and Tetsuya Ichikawa. Random Switching Logic: A Countermeasure against DPA based on Transition Probability. Cryptology ePrint Archive (`http://eprint.iacr.org/`), Report 2004/346, 2004.

[TK04] Elena Trichina and Lesya Korkishko. Secure and efficient aes software implementation for smart cards. In Chae Hoon Lim and Moti Yung, editors, *Information Security Applications, 5th International Workshop, WISA 2004, Jeju Island, Korea, August 23-25, 2004, Revised Selected Papers*, volume 3325 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2004.

[TSG03] Elena Trichina, Domenico De Seta, and Lucia Germani. Simplified Adaptive Multiplicative Masking for AES. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2535 of *Lecture Notes in Computer Science*, pages 187–197. Springer, 2003.

[WOL02] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. An ASIC implementation of the AES SBoxes. In Bart Preneel, editor, *Topics in Cryptology - CT-RSA 2002, The Cryptographer's Track at the RSA Conference 2002, San Jose, CA, USA, February 18-22, 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2002.