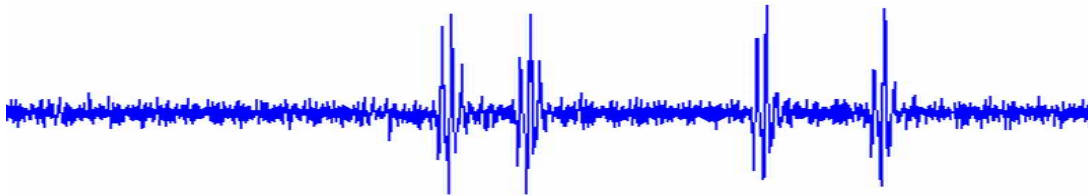
Technische Universität Graz
Institut for Applied Information Processing and Communications – IAIK
Inffeldgasse 16a
A–8010 GRAZ

IAIK
TUG

IAIK
TU Graz

SCA-Lab Technical Report Series



Secure and Efficient Masking of AES – A Mission Impossible?

Version 1.0

Elisabeth Oswald, Stefan Mangard and Norbert Pramstaller

Technical Report
IAIK - TR 2003/11/1 (updated on 2004/06/04)

http://www.iaik.tu-graz.ac.at/research/sca-lab/index.php

Secure and Efficient Masking of AES – A Mission Impossible?

Elisabeth Oswald, Stefan Mangard and Norbert Pramstaller

4th June 2004

Abstract

This document discusses masking approaches with a special focus on the AES S-box. Firstly, we discuss previously presented masking schemes with respect to their security and implementation. We conclude that algorithmic countermeasures to secure the AES algorithm against side-channel attacks have not been resistant against all first-order side-channel attacks. In this article, we introduce a new masking countermeasure which is not only secure against first-order side-channel attacks, but which also leads to relatively small implementations compared to other masking schemes when implemented in dedicated hardware.
Keywords: AES, Masking, Side-Channel Attacks

1 Masking Approaches

In the following, we discuss the different approaches that have been proposed in order to mask implementations of the Advanced Encryption Standard (AES)[Nat01]. The AES is a byte oriented cipher, *i.e.*, the basic data blocks are 8-bit blocks. Masking AES means masking the intermediate bytes which are processed in an AES computation. Masking a byte value x means to choose a random value m (the *mask*) and to define a function f (the *masking*) which takes both values as input to calculate the masked output: $f(x,m) = x \star m$. The operator $\star$ is either defined as bit-wise Xor operation, denoted by $+$, (*additive masking*), or as multiplication, denoted by $\times$, over a finite field (*multiplicative masking*).

AES consists of four transformations, **SubBytes(), ShiftRows(), MixColumns()** and **AddRoundKey()**. Except for **SubBytes()**, all transformations are linear, *i.e.*, they have the property that $f(x+m) = f(x) + f(m)$. Hence, for such transformations, it is an easy task to compute how the mask m has changed during the transformation. As a consequence, it is also simple to re-establish the original mask m after a (sequence of) transformation(s). More difficult is the transformation **SubBytes()**. As **SubBytes()** is non-linear, re-establishing the mask after a **SubBytes()** transformation is difficult.. This is because

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

In the following subsections, we review the different methods that can be used to mask **SubBytes()**.

1.1 Re-computation of SubBytes()

The **SubBytes()** transformation can be implemented as a table lookup. Suppose we want to define a masking such that

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

In order to achieve this with a table lookup we have to compute a corresponding table **MaskedSubBytes()** for the mask m.

Algorithm 1 Computation of Masked **SubBytes()**

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

Such a table needs to be computed for each mask value m_i. If i different masks m are used, then the complexity of this procedure is $i * 256$. If we ensure, that the same masks m_i are re-established before the **SubBytes()** operation in each round, the same i tables can be used throughout the complete AES calculation.

1.2 Multiplicative Masking

Especially in a dedicated hardware implementation, tables which are implemented as ROMs are rather area expensive. In AES, **SubBytes()** can be implemented very efficiently using composite field arithmetic [WOL02]. Hence, the computation of masked tables is not an attractive option. Another idea, presented in [AG01], is based on using multiplicative masks. The **SubBytes()** transformation is defined as

$$\mathbf{SubBytes}(\mathbf{x}) = A \times x^{-1} + b.$$

In this definition, A is a fixed 8×8 matrix, and b is a column vector of length 8. The inversion operation itself, denoted by $Inv(x) = x^{-1}$ is calculated over the finite field $GF(256)$. This operation is compatible with the multiplication over the finite field:

$$Inv(x \times y) = Inv(x) \times Inv(y).$$

Using this property, it is an easy task to remove the additive mask m and replace it with a multiplicative mask $m' \neq 0$ and calculating the inversion without ever revealing the value x.

Algorithm 2 Multiplicative Masking of **SubBytes()**

INPUT: $x + m$, m,m'
OUTPUT: $Inv(x) + m$,

1: $(x + m)$ $/ \times m'$
2: $(x + m) \times m'$ $/ + (m \times m')$
3: $x \times m'$ $/^{-1}$
4: $x^{-1} \times m'^{-1}$ $/ + (m \times m'^{-1})$
5: $(x^{-1} \times m'^{-1}) + (m \times m^{-1})$ $/ \times m'$
6: $x^{-1} + m$

Algorithm 2 requires 4 multiplications, 1 inversion and 2 Xors in addition to the original inversion. This additional effort can be reduced to 2 multiplications, 1 squaring, 1 Xor and one Xor with 1, by taking $m' = m$.

In AES, an **AddRoundKey** operation is performed prior to the first encryption round and thus, prior to the first time when the inversion needs to be computed. If a key byte k equals a data byte d, then the result of **AddRoundKey**, which is $x = d + k$, equals zero. This observation can readily be used in an attack which is referred to as **zero-value attack** and was introduced in [GT03]. Let t denote a power measurement (trace) and let the set of all traces t be denoted by T. Suppose a number of AES encryptions is executed and their power consumption is measured. Assume that the input texts are known. For all 256 possible key-bytes k', we do the following. We define a set M_1 which contains those measurements with $k' = d$ right before the **SubBytes()** transformation. We also define a set M_2 which contains the measurements with $k' \neq d$ right before the **SubBytes()** transformation.

$$M_1 = \{t \in T : k' = d\} \tag{1}$$
$$M_2 = \{t \in T : k' \neq d\} \tag{2}$$

If $k = k'$, then the two sets M_1 and M_2 must show a considerable difference at the point in time when the masked **SubBytes()** operation has been performed. This is due to the fact that set M_1 contains the measurements in which the 0-value is manipulated in the inversion. If $k \neq k'$, then the definition of the sets is meaningless. Hence, no difference between the sets can be observed. The difficulty in this scenario is that one needs enough traces in M_1 in order to get rid of noise. As, on average, the probability of $k' = d$ is $\frac{1}{256}$, a considerable amount of measurements have to be acquired (see Appendix B for a thorough statistical analysis).

1.3 Simplified Multiplicative Masking

This was presented in [TSG03]. It is equivalent to the original multiplicative masking scheme, but sets $m = m'$. Hence we get Algorithm 3.

Algorithm 3 Simplified Multiplicative Masking of **SubBytes()**

INPUT: $x + m$, m
OUTPUT: $Inv(x) + m$,

1: $(x + m)$ $\quad / \times m$
2: $(x + m) \times m$ $\quad / + (m \times m)$
3: $x \times m$ $\quad /^{-1}$
4: $x^{-1} \times m^{-1}$ $\quad / + 1$
5: $(x^{-1} \times m^{-1}) + 1$ $\quad / \times m$
6: $x^{-1} + m$

In Algorithm 3, we do not need to compute the inverse of mask, and we require one less multiplication. Of course, we are not longer allowed to have $m = 0$. It is also not clear, what implications it has (at least for software implementations) if an attacker is able to monitor $x + m$ and $x * m$.

1.4 Embedded Multiplicative Masking

In order to overcome the problem with zero-values of the multiplicative masking, the so-called embedded multiplicative masking was suggested in [GT03]. The key idea of this masking is to embed the finite field in which the inversion operation is defined, *i.e.*, $GF(2)[x]/P(x)$, and $P(x) = (x^8 + x^4 + x^3 + x + 1)$, into the larger ring $R = GF(2)[x]/(P(x)*Q(x))$. The polynomial $Q(x)$ has degree k and needs to be irreducible over $GF(2)$ and co-prime to $P(x)$. To repair the multiplicative masking, a random embedding $\rho : GF(2)[x]/P(x) \rightarrow \mathbf{R}$ is defined as :

$$\rho(U) = U + R \times P \tag{3}$$

In Equation 3, a random polynomial $R(x) \neq 0$ of degree smaller than k is used as a mask. If U is an additively masked value, *i.e.*, $U = x + m$, then the additive mask m can be removed after the computation of ρ. In order to calculate the inversion in $GF(2)[x]/P(x)$, we define a mapping F on $\mathbf{R}$ which coincides with the inversion on $GF(2)[x]/P(x)$. This mapping is $F : \mathbf{R} \rightarrow \mathbf{R}$ with $F(U) = U^{254}$. Reducing $F(U)$ modulo $P(x)$ gives the inverse of U in $GF(2)[x]/P(x)$. The additive masking can be restored before performing this final reduction step. If the polynomial R is indeed random, the masking will not allow to reveal exploitable information even in the case where $x = 0$.

1.5 Combinational Logic Design for AES Subbyte

The masking schemes, which we have discussed so far, are all algorithmic countermeasures and can be implemented either in software or in hardware. However, masking on gate level, *i.e.*, using a special masked logic, has been proposed as well. In [Tri03], this idea has been picked up and applied specifically for AES. The principle, which is presented in [Tri03], is similar to what we will propose in the next section of this article. The authors of [Tri03]

have also observed that in order to counter zero-value attacks, all intermediate values must be concealed by an additive mask. As we have already explained, the only operations, which are tricky to mask occur during the computation of the AES S-box. In particular, if the S-box is implemented in composite field arithmetic, it is the AND gate, which is difficult to mask. The main contribution of [Tri03] is therefore the design of a masked AND gate and its clever use to secure implementations of the AES S-box.

However, in comparison to our approach, there are several major differences which we highlight in the subsequent paragraphs.

The countermeasure of [Tri03] works on gate level and therefore it can not be implemented efficiently in software on an arbitrary processor. Another major disadvantage of this approach is that masking on gate level is covered by several patents, for example [MD01], [KKG02] and [MP03].

In contrary, our countermeasure works on the algorithmic level and can be implemented in both hard- and software. Consequently, our proposal does not require masked gates and our technique is not covered by patents (to our best knowledge).

2 Combined Masking in Tower Fields

In order to thwart zero-value attacks, we have developed a new scheme which works with combinations of additive and multiplicative masks. Throughout the whole cipher, including the S-box computation, the data is concealed by an additive mask.

Before going into the details of the new scheme, we review some facts about efficient implementations of the inversion operation first.

2.1 Inversion in $GF(256)$

Our S-box design follows the architecture proposed in [WOL02] which is based on composite field arithmetic. In this approach, each element of $GF(256)$ is represented as a linear polynomial $a_h x + a_l$ over $GF(16)$. The inversion of such a polynomial can be computed using only operations in $GF(16)$:

$$(a_h x + a_l)^{-1} = a'_h x + a'_l \quad (4)$$

$$a'_h = a_h \times d^{-1} \quad (5)$$

$$a'_l = (a_h + a_l) \times d^{-1} \quad (6)$$

$$d = ((a_h^2 \times p_0) + (a_h \times a_l) + a_l^2). \quad (7)$$

The element p_0 is defined in accordance with the field polynomial which is used to define the quadratic extension of $GF(16)$, see [WOL02].

The finite fields which we will frequently use in the following sections are:

$$\begin{aligned} GF(256) &\simeq GF(2)[x]/(x^8+x^4+x^3+x+1) \\ GF(16) &\simeq GF(2)[x]/(x^4+x+1) \\ GF(4) &\simeq GF(2)[x]/(x^2+x+1). \end{aligned}$$

2.2 Masked Inversion in $GF(256)$

In our masking scheme for the inversion, all intermediate values as well as the input and the output are masked additively. In order to calculate the inversion of a masked value input value, we first map the value as well as the mask to the composite field representation shown in [WOL02]. This mapping is a linear operation and therefore it is easy to mask. After the mapping, the value that needs to be inverted is represented by $(a_h + m_h)x + (a_l + m_l)$ instead of $a_h x + a_l$. Both values, a_h and a_l, are masked additively.

Our goal is to achieve that all input and output values in Equation 4 are masked, by using only masked values in combination with appropriate correction terms:

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

Suppose we would calculate Equation 5 with masked input values,*i.e.*, with $a_h + m_h$ instead of a_h and with $d^{-1} + m_l$ instead of d^{-1}. Then we would get the following result:

$$(a_h + m_h) \times (d^{-1} + m_l) = a_h \times d^{-1} + m_h \times d^{-1} + a_h \times m_l + m_h \times m_l.$$

Apparently, this is not equal to $a'_h + m_h = a_h \times d^{-1} + m_h$, because the masks m_h and m_l introduce several additional terms. We have to get rid of these additional terms in order to establish the correct result. The correction terms which have to be added are $(d^{-1} + m_l) \times m_h$, $(a_h + m_h) \times m_l$, $m_h \times m_l$ and m_h. One has to take care when adding those terms that no intermediate value is correlated with hypotheses, *i.e.*, values which an attacker can predict.

In order to achieve this secure application of correction terms, we use a fresh mask M, which masks the summation of the correction terms. M needs to be independent from m_h and m_l. Details about the computation of the correction terms for Equations 5-7 can be found in the subsequent sections.

The remaining issue that needs to be resolved is the secure computation of the masked inversion in $GF(16)$. So far, we have only shifted our problem from the bigger field down to the smaller field. However, there is no way to totally avoid the computation of the masked inversion. Hence, we need to find an algebraic structure in which the inversion operation is a linear operation. Although this seems to be intuitively impossible, there is a nice solution. In the finite field with four elements, $GF(4)$, the inversion operation is indeed a linear operation. Fortunately, $GF(16)$ is a quadratic extension of it, *i.e.* $GF(16) \simeq GF(4) \times GF(4)$, and $GF(256)$ is a quadratic extension of $GF(16)$, *i.e.*, $GF(256) \simeq GF(16) \times GF(16)$. Hence, $GF(256)$ is a tower field over $GF(4)$ and this is the property that we exploit.

We always use additively masked values to calculate the inversion operation according to Equations 5-7. We have to add appropriate correction terms in order to achieve the correct result of these equations. Because $GF(256)$ is a tower field over $GF(4)$, we can shift the computation of the masked inversion, which needs to be done for d, down to $GF(4)$. In this field, the inversion is a linear operation and thus, it is easy to mask. In the remainder of this section, we provide the details for the secure calculation of the Equations 5-7.

2.3 Secure Calculation of the Masked d

We have to calculate a masked d, which we call $d' = d + m_h$, according to Equation 7 based on the input data $(a_h + m_h)x + (a_l + m_l)$ and M (a fresh mask). Note that M is just used to mask the addition of the correction terms—it is not part of the final result.

The terms dm_1, dm_2 and dm_3 are the result of filling in masked input values into Equation 7. In order to get a correctly masked value d, the correction terms $c_1 \ldots c_6$ have to be added. The calculation of these correction terms c_i requires 3 multiplications, 2 squarings and 1 multiplication with a constant term in $GF(16)$.

$$
\begin{aligned}
d' = d + m_h = \\
&= \underbrace{(a_h + m_h)^2 \times p_0}_{dm_1} + \mathbf{M} + \underbrace{(a_h + m_h) \times (a_l + m_l)}_{dm_2} + \underbrace{(a_l + m_l)^2}_{dm_3} \\
&+ \underbrace{(\mathbf{a_h} + \mathbf{m_h}) \times \mathbf{m_l}}_{c_1} + \underbrace{(\mathbf{a_l} + \mathbf{m_l}) \times \mathbf{m_h}}_{c_2} \\
&+ \underbrace{\overbrace{\mathbf{m_h^2}}^{c_3'} \times \mathbf{p_0}}_{c_3} + \underbrace{\mathbf{m_l^2}}_{c_4} \\
&+ \underbrace{\mathbf{m_h} \times \mathbf{m_l}}_{c_5} + \underbrace{\mathbf{m_h}}_{c_6} + \mathbf{M}
\end{aligned}
\tag{8}
$$

2.4 Secure Calculation of the Masked a'_h

The coefficient a'_h in Equation 5 also needs to be calculated with masked input values. Hence, we need to work with $a_h + m_h$, M and $d'^{-1} = d^{-1} + m_l$:

$$
\begin{aligned}
a'_h + m_h = &\underbrace{(a_h + m_h) \times (d^{-1} + m_l)}_{dm_4} + \mathbf{M} \\
&+ \underbrace{(\mathbf{d^{-1}} + \mathbf{m_l}) \times \mathbf{m_h}}_{c_7} + \underbrace{(\mathbf{a_h} + \mathbf{m_h}) \times \mathbf{m_l}}_{c_1} \\
&+ \underbrace{\mathbf{m_h}}_{c_6} + \underbrace{\mathbf{m_h} \times \mathbf{m_l}}_{c_5} + \mathbf{M}
\end{aligned}
\tag{9}
$$

The value d_m that is calculated by filling in masked values into Equation 5, needs to be corrected as shown in equation 9. The calculation of the correction term c_7 requires 1 multiplication in $GF(16)$. All other correction terms can be reused from Equation 8.

2.5 Secure Calculation of the Masked a'_l

We use $a_l + m_l$, M and $d'^{-1} = d^{-1} + m_h$ to calculate the masked value a'_l as shown in Equation 10.

$$\begin{aligned} a'_l + m'_l = (a_h \times d^{-1} + m_h) + \underbrace{(a_l + m_l) \times (d^{-1} + m_h)}_{dm_5} + \mathbf{M} \\ + \underbrace{(\mathbf{d^{-1}} + \mathbf{m_h}) \times \mathbf{m_l}}_{c_8} + \underbrace{(\mathbf{a_l} + \mathbf{m_l}) \times \mathbf{m_h}}_{c_2} \\ + \underbrace{\mathbf{m_l}}_{c_9} + \underbrace{\mathbf{m_h}}_{c_6} + \underbrace{\mathbf{m_l} \times \mathbf{m_h}}_{c_5} + \mathbf{M} \end{aligned} \tag{10}$$

The calculation of the correction terms c_8 requires 1 multiplication in $GF(16)$. All other correction terms can be reused from the previous computations.

2.6 Secure Calculation of the Masked d^{-1} in $GF(4) \times GF(4)$

It is necessary to calculate the inverse of d in $GF(16)$ for Equations 5 and 6.

As explained in Section 2.2, calculating this inverse can be reduced to calculating the inverse in $GF(4)$ by representing $GF(16)$ as quadratic extension of $GF(4)$. Details of this representation and the transformation matrix to map elements of one representation to the other are given in Appendix A.

An element of $GF(4) \times GF(4)$ is a linear polynomial with coefficients in $GF(4)$, *i.e.*, $a = (a_h \times x + a_l)$, with a_h and $a_l \in GF(4)$. The same formulae as given in Equations 4-7 can be used to calculate the inverse in $GF(4) \times GF(4)$. In $GF(4)$, the inversion operation is equivalent to squaring: $x^{-1} = x^2 \, \forall x \in GF(4)$. Hence, in $GF(4)$ we have that $(x + m)^{-1} = (x + m)^2 = x^2 + m^2$; the inversion operation preserves the masking in this field.

The remaining operations for the calculation of the inversion in $GF(4) \times GF(4)$ is done according to Equations 4-7. Note that the mask of the inverted d equals m_h^2. In order to get $d^{-1} + m_h$ as final result, we also need to change from this mask to m_h by adding first a fresh mask M and then removing the old mask by adding $m_h^2 + m_h + M$.

3 Practical Implementation of the New Masking Method

We have implemented our new masked S-box (which we call **SubBytes NEW**) using a 0.25 μm CMOS technology with 5 metal layers. We have compared our implementation to implementations of the schemes which are described in Section 1.2 (**SubBytes Akkar**), see Figure 1 and in Section 1.3 (**SubBytes Trichina**), see Figure 2.

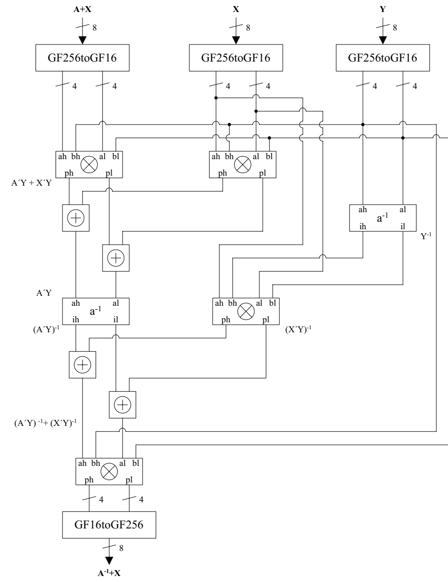


Figure 1: Gate-level implementation of Algorithm 2.

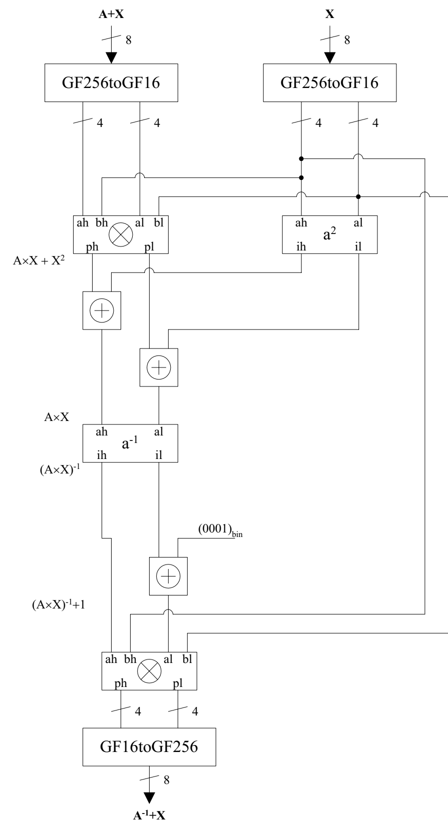


Figure 2: Gate-level implementation of Algorithm 3.

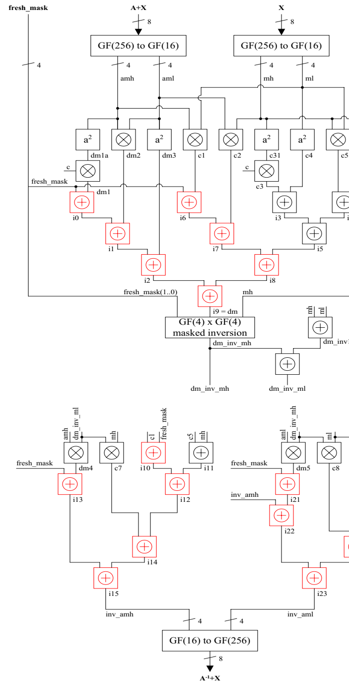


Figure 3: Gate-level implementation of the inversion in $GF(16) \times GF(16)$.

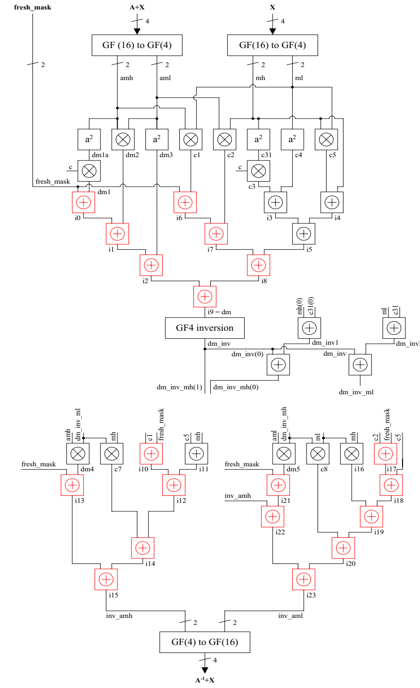


Figure 4: Gate-level implementation of the inversion in $GF(4) \times GF(4)$.

Furthermore we have compared our new, masked S-box with an implementation of an S-box that is based on composite field arithmetic as described in [WOL02] (**SubBytes Wolkerstorfer**).

3.1 Secure Implementation

An unskilled implementation of the S-box on a lower level (like gate-level for example), can render our effort useless. It is mandatory to sum up the intermediate values which occur in the new S-box in a way such that masks cannot cancel each other out. In addition, timely glitches need to be considered as well.

Figures 3 and 4 show how the algorithmic description of the masked S-box has been mapped to a gate-level description. The critical XOR operations, which add the intermediate values, are depicted in red color in these figures.

3.2 Comparison of the Implementations

The implementation of **SubBytes Akkar** and **SubBytes Trichina** make use of multiplications in finite fields. We decided to base the multipliers for that operation on the

same, optimized multipliers which we used for the implementation of **SubBytes Wolkerstorfer**. Because **SubBytes Wolkerstorfer** uses composite field arithmetic, three $GF(16)$ multipliers and one $GF(16)$ constant coefficient multiplier had to be combined according to [Paa94] to build a $GF(2)[x]/(x^8+x^4+x^3+x+1)$ multiplier. The implementations compare with respect to their area-time (AT) product as shown in Figure 5.

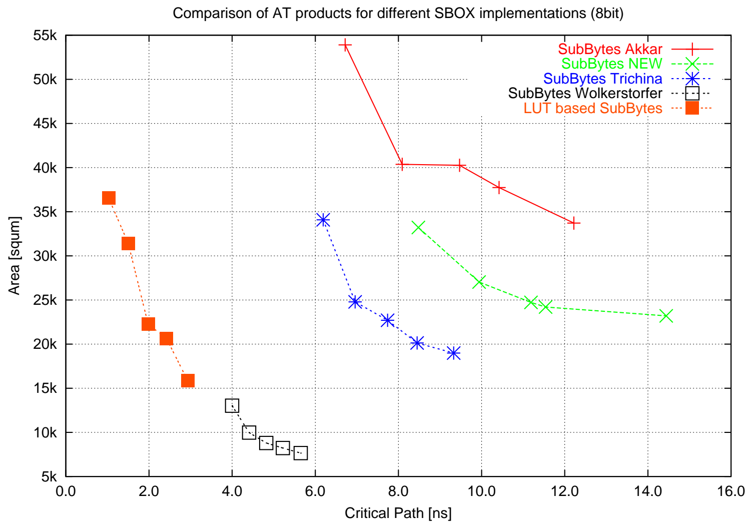


Figure 5: Comparison of different S-box implementations

As shown in Figure 5, our new S-box is better than **SubBytes Akkar** but less efficient than **SubBytes Trichina**. However, our S-box has the the advantage of being secure, while **SubBytes Trichina** is susceptible to zero-value attacks as well as standard DSCA attacks [ABG04].

3.3 Practical Realization

As mentioned in Section 3, we have made an implementation in dedicated hardware. This implementation was made in co-operation with the ETH Zurich, during the master thesis research (winter term 2003/2004) of Norbert Pramstaller, who was an exchange student at the ETH at that time. The resulting chip, named ARES, has been manufactured. It is beeing functionally verified at the moment, and will be ready for side channel analysis in the next month. An article describing the design of ARES has been accepted at the ESSCIRC 2004 conference [PGH+04].

4 Conclusions

We have presented a new masking scheme to secure AES. This masking scheme uses a combination of additive and multiplicative masks and computes the inversion in $GF(4)$.

Additive masks provide security. In combination with multiplicative masks we achieve security and efficiency. Very important is that we shift the inversion operation down to $GF(4)$ where it is a linear operation. It is therefore easy to mask.

We have also compared the area-time product of implementations of two other masking schemes ([AG01] and [TSG03]) with an implementation of our new design. It has turned out that our algorithm is better than [AG01] and slightly worse than [TSG03] with respect to the area-time product. In contrast to both schemes, our algorithm does not succumb to zero-value attacks.

References

[ABG04] M.-L. Akkar, R. Bevan, and L. Goubin. Two Power Analysis Attacks against One-Mask Methods. In *FSE 2004 – Pre-proceedings*, pages 308–325, 2004.

[AG01] M.-L. Akkar and C. 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.

[GT03] J. D. Golić and Ch. 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 (LNCS)*, pages 198–212. Springer, 2003.

[KKG02] Franz Klug, Oliver Kniffler, and Berndt Gammel. Rechenwerk, Verfahren zum Ausführen einer Operation mit einem verschlüsselten Operanden, Carry-Select-Addierer und Kryptographieprozessor. German Patent DE 10201449 C1, January 2002.

[MD01] T. S. Messerges and E. A. Dabbish. Method of Preventing Power Analysis Attacks on Microelectronic Assemblies. United States Patent, Patent No.: US 6,298,135 B1, October 2001.

[MP03] R. Menicocci and J. Pascal. Elaborazione crittografica di dati digitali mascherati. Italian Patent Application MI2003A001375, July 2003.

[Nat01] National Institute of Standards and Technology. FIPS-197: Advanced Encryption Standard, November 2001.

[Paa94] Ch. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, Institute for Experimental Mathematics, University of Essen, 1994.

[PGH+04] N. Pramstaller, F.K. Gürkaynak, S. Haene, H. Kaeslin, N. Felber, and W. Fichtner. Towards an AES Crypto-chip Resistant to Differential Power Analysis. In *ESSCIRC 2004 Conference Proceedings*, 2004. to be published.

[Tri03] E. Trichina. Combinational Logic Design for AES SubByte Transformation on Masked Data. Cryptology ePrint Archive, Report 2003/236, 2003. `http://eprint.iacr.org/`.

[TSG03] E. Trichina, D. De Seta, and L. 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 (LNCS)*, pages 187–197. Springer, 2003.

[WOL02] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES SBoxes. In *Cryptographer's Track at the RSA Conference 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2002.

A Composite Fields – Mappings and Arithmetic

A binary finite field $GF(2^k)$ with $k = n*m$ can also be represented as a so-called composite field $GF(2^n)^m)$. There are several representations for both fields. Hence, there are several mappings from one representation to the other. In the following sections, we explain how to compute this mappings [Paa94]. Furthermore, we point out the choices one has, and finally, we select the mapping that we consider best in the sense that it minimizes the number of (subsequent) Xor gates.

A.1 The finite field $GF((2^4)^2)$

The finite field with 256 elements, $GF(256)$ or $GF(2^8)$, can be seen as a quadratic extension of the finite field with 16 elements, $GF(2^4) = GF(16)$. Hence, each element of $GF(256)$ can be represented as a 2-dimensional vector, or a linear polynomial over $GF(16)$: $a = (a_h \times x + a_l), a \in GF((2^4)^2)$, $a_h \in GF(2^4)$ and $a_l \in GF(2^4)$. We can define an isomorphic mapping of the field elements which are represented with respect to $GF(2^8)$ to element which are represented with respect to $GF(2^4)$. Before we explain how to compute the mapping in Mathematica, we define some notation first. The field polynomial of $GF(256)$ is denoted by $R(z)$. The field polynomial of $GF(16)$is denoted by $Q(y)$ and the field polynomial of $GF((2^4)^2)$ is denoted by $P(x)$. Because only the field polynomial of $GF(256)$ was defined, we could choose $Q(y)$ (which we set to $y^4 + y + 1$ as indicated before) and $P(x)$. The polynomial $P(x)$ has coefficients in $GF(2^4)$, hence, the more coefficients have a value $\neq 0, 1$, the more complicated the arithmetic will be. Logically, we searched for a polynomial of the form $x^2 + x + c$, where c is a suitable constant such that the polynomial itself is

primitive. Several of such polynomials have been found. The best, with respect to the mapping and the arithmetic is the one with $c = \omega^{11}$, and ω is a primitive root of $GF(2^4)$.

In order to construct the mapping, we are looking for the 8 base elements of $GF((2^4)^2)$ to which the 8 base elements of $GF(2^8)$ are to be mapped. The one element must be mapped to the one element. The base element β of $GF(2^8)$ is mapped to some base element α^t of $GF((2^4)^2)$, the base element β^2 to α^{2t}, and so on. The mapping must be homomorphic with respect to both field operations. Hence, we need to map β to an α^t such that

$$R(\alpha^t) = 0 \pmod{Q(y), P(x)} \tag{11}$$

holds.

The mapping that we chose, and the resulting finite field arithmetic is described in [WOL02].

A.2 The finite field $GF((2^2)^2)$

In the same way, as described in Section A.1, we can represent $GF(2^4)$ as $GF((2^2)^2)$. Again, The field polynomial of $GF(16)$ is denoted by $Q(y)$. The field polynomial of $GF(4)$ is denoted by $F_4(u)$ and the field polynomial of $GF(4)\times GF(4)$ is denoted by $F_{4x4}(v)$. Because only the field polynomial of $GF(16)$ is already defined in [WOL02], we may choose $F_4(u)$ and $F_{4x4}(v)$. The polynomial $F_{4x4}(v)$ has coefficients in $GF(4)$, hence, the more coefficients have a value $\neq 0, 1$, the more complicated the arithmetic will be. Logically, we searched for a polynomial of the form v^2+v+c, where c is a suitable constant such that the polynomial itself is primitive.

The field polynomials which we selected are:

$$F_4(u) = u^2 + u + 1, F_4(\gamma) = 0 \tag{12}$$

$$F_{4x4}(v) = v^2 + v + \gamma^2, F_{4x4}(\delta) = 0. \tag{13}$$

In order to construct the mapping, we are looking for the 4 base elements of $GF(4) \times GF(4)$ to which the 4 base elements of $GF(16)$ are to be mapped. The one element,*i.e.*, the neutral element with respect to the multiplication, must be mapped to the one element. The base element δ of $GF(16)$ is mapped to some base element γ^t of $GF(4) \times GF(4)$, the base element δ^2 to γ^{2t}, and so on. The mapping must be homomorphic with respect to both field operations. Hence, we need to map δ to an γ^t such that

$$Q(\delta^t) = 0 \pmod{F_4(u), F_{4x4}(v)} \tag{14}$$

holds true.

The four powers t that satisfy Equation 14 are $2, 4, 8, 16$. We chose $t = 2$, and computed the mapping and the inverse mapping

$$\mathbf{T_{4x4}} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{T_{4x4}^{-1}} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Arithmetic in $GF(2^2)$ can be done very efficiently. The bit-level definition of the multiplication of two values $a = (a_0, a_1)$ and $b = (b_0, b_1)$, $c = (c_0, c_1) = a \times b$, is given by:

$$h = a_1 b_1 \tag{15}$$
$$c_0 = a_0 b_0 + h \tag{16}$$
$$c_1 = a_1 b_0 + a_0 b_1 + h. \tag{17}$$

Squaring an element $a = (a_0, a_1)$, $c = a \times a$, which is equivalent to computing the inverse of a, a^1, is given by

$$c_0 = a_1 + a_0 \tag{18}$$
$$c_1 = a_1. \tag{19}$$

The multiplication with the constant γ^2, $c = \gamma^2 \times a$ is given by

$$c_0 = a_1 \tag{20}$$
$$c_1 = a_0 + a_1 \tag{21}$$

B Zero Value Attacks – A Statistical Analysis

Zero-value attacks are based on the fact that the value 0 can not be masked in a multiplicative way. Multiplying the value 0 with a random mask always leads to 0, which is the unmasked value.

In this section, we analyze the effectiveness of a zero-value attack. For this purpose, we assume that the attacker performs a classical first-order DPA attack on an implementation that uses multiplicative masking.

Like in a any classical DPA attack, hypotheses are formulated about the key. Based on these hypotheses, the attacker splits the power traces into two groups. We call these groups *high* and *low*, respectively. The attacker calculates the mean traces of the two groups. These mean traces are of course a function of the time. However, in order to make the following equations more readable, we only consider the moment of time of the mean traces, where the biggest difference between them occurs.

We model the power consumption at this moment of time in the following way: $Power = S + \mathcal{N}(\mu, \sigma)$. S is the power consumption that is caused by the processing of the attacked intermediate result. This signal S is buried in gaussian noise with the mean μ and the variance σ. Taking the mean of n power traces leads to a reduction of the variance to $\frac{\sigma}{\sqrt{n}}$.

Therefore, we can write the means M_h and M_l as shown in the equations 22 and 23.

$$M_h = S_h + \mathcal{N}\left(\mu_h, \frac{\sigma_h}{\sqrt{n_h}}\right) \tag{22}$$

$$M_l = S_l + \mathcal{N}\left(\mu_l, \frac{\sigma_l}{\sqrt{n_l}}\right) \tag{23}$$

$$\tag{24}$$

We assume that the distributions $\mathcal{N}(\mu_h, \sigma_h)$ and $\mathcal{N}(\mu_l, \sigma_l)$ are independent. Therefore, the difference of the means $M_d = M_h - M_l$ can be calculated as shown in equation 25.

$$M_d = M_h - M_l = S_h - S_l + \mathcal{N}\left(\mu_h - \mu_l, \sqrt{\frac{{\sigma_h}^2}{n_h} + \frac{{\sigma_l}^2}{n_l}}\right) \tag{25}$$

In a classical attack, the size and the variance of the two groups *high* and *low* is approximately equal. Therefore, we can make the following simplifications: $n_h = n_l = \frac{n}{2}$ and $\sigma = \sigma_h = \sigma_l$. This leads to equation 26. n corresponds to the total number of samples.

$$M_d = M_h - M_l = S_h - S_l + \mathcal{N}\left(\mu_h - \mu_l, \sqrt{\frac{4\sigma^2}{n}}\right) \tag{26}$$

In a zero-value attack, the two groups *high* and *low* do not have the same size. Assuming that the input of the encryption is uniformly distributed, the following relation holds true: $n_h = \frac{n}{256} n_l = \frac{255n}{256}$. Like in the previous case, we additionally assume that $\sigma = \sigma_h = \sigma_l$. This leads to equation 27.

$$M_d = M_h - M_l = S_h - S_l + \mathcal{N}\left(\mu_h - \mu_l, \sqrt{\frac{65536\sigma^2}{255n}}\right) \tag{27}$$

Obviously, the variance of M_d is much bigger in a zero-value attack than in a classical DPA attack. In the following subsection, we analyze how the number of needed samples depends on the variance of M_d.

B.1 The Relationship of the Variance to the Number of Samples

In this subsection, we show how the number of samples needed in a DPA attack depends on the variance of the difference distribution.

The goal of the attacker performing a DPA attack is to detect a significant peak in the difference trace. This means that the probability $p(M_d > 0)$ should be high. The higher this probability is, the more significant is the peak that the attacker sees. If the probability is 0.5, there will be no significant peak.

In the following equations, μ_d is the mean of the difference distribution and σ_d is the variance of the difference distribution. The probability that M_d is below a certain threshold b can be calculated as shown in equation 28.

$$p(M_d < b) = \Phi\left(\frac{b - \mu_d}{\sigma_d}\right) \tag{28}$$

The probability that M_d is bigger than 0 can be calculated as shown in equation 29.

$$p(M_d > 0) = 1 - \Phi\left(\frac{0 - \mu_d}{\sigma_d}\right) \tag{29}$$

Based on the equation $\Phi(-M_d) = 1 - \Phi(M_d)$, equation 29 can be transformed to equation 30.

$$p(M_d > 0) = \alpha = \Phi\left(\frac{\mu_d}{\sigma_d}\right) \tag{30}$$

Using the property $\alpha = \Phi(Z_\alpha)$, we can formulate an equation that shows the relation between the quantile Z_α and μ_d as well as σ_d.

$$Z_\alpha = \frac{\mu_d}{\sigma_d} \tag{31}$$

Two attacks require the same amount of samples, if $p(M_d > 0) = \alpha$ is equal in both attacks. In order to determine how many samples more are needed in an attack A than in an attack B, we determine f in equation 32.

$$Z_\alpha = \frac{\mu_a}{\frac{\sigma_a}{\sqrt{n}}} = \frac{\mu_b}{\frac{\sigma_b}{\sqrt{f*n}}} \tag{32}$$

This leads to equation 33:

$$f = \frac{\mu_a^2}{\mu_b^2}\frac{\sigma_b^2}{\sigma_a^2} \tag{33}$$

Filling in the values for a classical first-order DPA attack and those for a zero-value attack, we get $f = 64.25$.

$$f = \frac{\mu_a^2}{\mu_b^2}\frac{\sigma_a^2}{\sigma_b^2} = \frac{\frac{65536\sigma^2}{255n}}{\frac{4\sigma^2}{n}} = \frac{16384}{255} = 64.25 \tag{34}$$

A copy-and-paste example for Matlab:

```
mu_a = 1; sigma_a = 8; samples_a = 100;
mu_b= 2; sigma_b = 17;
samples_b = samples_a * mu_a^2/mu_b^2*sigma_b^2/sigma_a^2;

sum(mean(normrnd(mu_a,sigma_a,samples_a,10000)) > 0)
sum(mean(normrnd(mu_b,sigma_b,round(samples_b),10000)) > 0)
```

This example shows that in both cases the probability that a value bigger than 0 is drawn is the same.