Side-Channel Analysis Resistant Implementation of AES on Automotive Processors

Master Thesis
Ruhr-University Bochum

Chair for Embedded Security
Prof. Dr.-Ing. Christof Paar

from
Andreas Hoheisel
June 12, 2009

Co-Advised by:
Dipl.-Ing. Timo Kasper (Ruhr-University Bochum)
Dr. Torsten Schütze (Robert Bosch GmbH, CR/AEA)

Statement

I hereby declare, that the work presented in this master thesis is my own work and that to the best of my knowledge it is original, except where indicated by references to other authors.

Erklärung

Hiermit versichere ich, dass ich meine Master Thesis selber verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Bochum, den 12. Juni 2009

Andreas Hoheisel

Contents

Glossary

Notation	Description
ACNS	Applied Cryptography and Network Security Conference
AES	Advanced Encryption Standard
cc	Cycle Count
CMTF	Combined Masking in Tower Fields
CPU	Central Processing Unit
DPA	Differential Power Analysis
ECU	Electronic Control Units
GF	Galois Field
GPR	General Purpose Registers
IP	Integer Pipeline
JTAG	Joint Test Action Group
KAT	Known Answer Tests
LS	Load Store Pipeline
LUT	Lookup Table
MCDS	Multi-Core Debug System
MISRA	Motor Industry Software Reliability
MSB	Most Significant Bit
NIST	National Institute of Standards and Technology
NOP	No Operation
PC	Program Counter
PLL	Phase-Locked Loop
PMM	Perfectly Masked Multiplication
PMS	Perfectly Masked Squaring
RNG	Random Number Generator
ROM	Read Only Memory

Notation	Description
SCA	Side Channel Analysis
SPA	Simple Power Analysis
splint	Secure Programming Lint
SPN	Substitution-Permutation Network
TLU	Table lookup
TMM	Transformed Multiplicative Masking
UDE	Universal Debug Engine
VCO	Voltage Controlled Oscillator
XOR	exclusive OR

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Nowadays, security becomes more and more important in the automotive industry. Robert Bosch GmbH, Corporate Research supplies the necessary know-how and provides solutions for future applications in the business units.

Many functions in a modern car are realized with the help of software applications and the underlying electronics. This means in particular, that a growing part of the components in a car are realized through an easy to copy and manipulate software. Due to this fact, there is a growing need for an effective protection of these software applications to save the intellectual property on the one hand and to disclose manipulation, e. g., by tuning, on the other hand.

There are up to 80 electronic control units (ECU) built into a modern upper class car. These control units come from different suppliers, communicate over potentially insecure networks with each other and with the outside world respectively, e. g., with diagnostic equipment or in the near future Car2Car and Car2Infrastructure. As a consequence, these control units should authenticate each other and the communication data should be cryptographically protected against manipulation.

For several use cases in the automotive domain, e. g., tuning detection and sensor protection, cryptography constitutes a considerable part of the overall-security architecture. Due to their performance and compactness in implementations, symmetric encryption primitives are of special interest. These primitives can be used for attaining the following security goals:

- confidentiality: encryption of data,
- authenticity: challenge-response protocols,
- integrity: message authentication codes.

The State-of-the-Art symmetric encryption scheme is the Advanced Encryption Standard (AES), [FIP01]. This block cipher with key sizes of 128, 192, and 256 bits and a block size of 128 bits is remarkably fast and can be implemented very efficiently on a wide variety of platforms. It replaces the old Data Encryption Standard DES or, to be more precise, its variant Triple-DES.

A lot of proprietary mechanisms in the automotive environment are getting replaced by standardized security mechanisms. An example is here the changeover from the proprietary KeeLoq System[1], which is used for keyless entry in cars and garage doors, to modern challenge-response protocols based on AES[2].

1 http://www.microchip.com

2 Embedded AES-crypto engine for keyless entry systems, ESCAR2007.

1.2 Adversary Model and Assumptions

Ross Anderson and Markus Kuhn give a good classification of adversaries' knowledge, funding, and skills in [AK98]. They define the following three classes of attackers:

- **Class I (clever outsiders):** They are very intelligent but may have insufficient knowledge of the system. They may have access to only moderately sophisticated equipment. They often try to take advantage of an existing weakness in the system, rather than try to create one.
- **Class II (knowledgeable insiders):** They have substantial specialized technical education and experience. They have varying degrees of understanding of parts of the system but potential access to most of it. They have highly sophisticated tools and instruments for analysis.
- **Class III (funded organizations):** They are able to assemble teams of specialists with related and complementary skills backed by great funding resources. They can do in-depth analysis of the system, design sophisticated attacks, and use the most advanced analysis tools. They may use Class II Adversaries as part of the attack team.

We will focus on knowledgeable insiders, i. e., the cryptographic algorithm itself is known to the attack agent (no security by obscurity!). The attack agent may have access to equipment like oscilloscopes etc. that can be found in university labs. The specific implementation, e. g., the source code or an open sample device, is assumed to be *not available* to the attacker.

In terms of side-channel analysis resistance[3], this means that our implementation should be resistant against:

- timing attacks, and
- first-order differential power analysis (DPA) attacks.

Resistance against timing attacks implies that we assume an adversary having access to an exact simulation of the runtime of the implementation. Note that we *do not require* resistance against the new microarchitectural side-channel attacks like cache and branch predication attacks [AKS07, Ber05, OST06, NS06]. This does not mean that these attacks may not be relevant, it just means that we think that an attack agent has no or only little chance to install a spy process on the target processor and to measure the timing signals with the necessary precision.[4]

We assume that table lookups (TLU) have a constant runtime. For achieving timing analysis resistance, we need a key-independent runtime of the implementation. Therefore we will consider the effects of the 4-stage pipeline of the target processor, since they have a huge influence on the runtime.

Resistance against first-order differential power analysis (DPA) attacks means that neither a standard difference of means nor a correlation attack against the implementation will succeed

3 All attack details will be explained in later chapters.

4 A further, more practical reason is that we do not have yet the exact specification of next generation ECU processor. Thus, it doesn't make much sense to optimize the countermeasure to the very specific design of current processors.

with a sufficient number of measurement traces. We assume that the attacker has no access to a training device where he can input, e. g., random masks or keys, i. e., we effectively rule out template attacks [GLRP06, APSQ06, CRR02].

The goal is not to make the implementation secure against second or higher order attacks, i. e., the focus is put on simple masking schemes. Of course, very simple second order attacks, e. g., by using the same mask twice directly on the bus, have to be prevented.

1.3 Scope

Robert Bosch GmbH, Corporate Sector Research and Engineering investigates the suitability of side-channel resistant implementations of symmetric key primitives under the constraints of the embedded domain. In this thesis, we will survey and investigate several masking countermeasures described in the literature to protect an implementation of the Advanced Encryption Standard (AES) against first-order DPA attacks in terms of security and runtime.

A selected countermeasure will be implemented in C (and partly in assembly language) on a typical 32-bit automotive processor used in Embedded Control Units (ECUs), e. g., TriCore. The implementation shall be secured against timing attacks as well. Of special interest is an efficient implementation taking into account the requirements of embedded devices with respect to memory and performance. Implementations of AES without countermeasures against DPA are provided by Robert Bosch CR/AEA. The tasks, that are to be carried out are:

- to understand and to survey the relevant AES implementation options,
- to understand and to survey the most important side-channel attacks (DPA: difference of means + correlation power attack), timing attack,
- to survey countermeasures (it is not the goal to invent new countermeasures, just to describe what is available in the literature),
- to describe the different masking schemes in their memory usage (RAM and ROM), performance, and number of random bits,
- to select and to implement the most appropriate or a combination of methods,
- to implement AES efficiently and securely with a key length of 128 bits on a TriCore 1796[5] processor in C and, maybe, parts of it in Assembler.

All countermeasures considered in this master thesis will be algorithmic, i. e., software-based, countermeasures. Although many hardware countermeasures have been discussed in the literature, e. g., special logic styles such as Masked Dual Rail Pre-Charge Logic (MDPL), increasing the noise by special noise generators, etc., we will only discuss these countermeasures if they can be applied to software as well. However, we will survey a number of countermeasures which have been originally proposed for hardware implementations e. g., masking in tower fields and investigate if they can be applied efficiently in software implementations.

5 The TriCore 1796 processor is currently used in most Engine Control Units of Bosch, e.g., MEDC17. It is a typical 32-bit RISC processor with four pipeline stages. The maximum CPU frequency is 150 MHz.

For the masking countermeasures, we need a certain amount of random data. The generation of these random values is not part of this thesis. We assume that all random values are given to our program via a pointer to an array with sufficient numbers of the necessary entropy. For simulation the stream cipher Grain-80 [HJM05] is used as pseudo random number generator (RNG).

We will present a side-channel resistant AES implementation for the TriCore architecture, which is not available up to now. Therefore we will combine side-channel analysis (SCA) countermeasures.

In Chapter 2 we will give a short overview of AES and the existing implementation options. We will introduce some mathematical basics which are important to understand the upcoming implementations. We introduce a processor model, which will help us to select the right implementation option for our needs.

Chapter 3 gives an overview about known side-channels and possible attacks on it. The focus lies on timing analysis because this form of side-channel will be analyzed at the end of this thesis.

Chapter 4 describes countermeasures to prevent side-channel attacks. We analyze different masking schemes in terms of size and runtime and combine them with other countermeasures like randomization.

Chapter 5 is dedicated to the TriCore. We will describe the architecture and the assembler instructions which are used for the implementation.

Chapter 6 is focused on the implementation. We will discuss improvements for the already available implementations and based on the improvements build a protected AES implementation.

In Chapter 7 we will analyze the implemented AES versions in terms of their timing behavior.

Chapter 2

AES – Advanced Encryption Standard

2.1 Introduction

The *Advanced Encryption Standard* (AES), also known as Rijndael, is a symmetric block cipher that was developed by the Belgian cryptographers Joan Daemen and Vincent Rijmen. AES encrypts/decrypts data with a block size of 128 bits and a key size of 128, 192, or 256 bits. AES targets on a small and fast implementation in hardware and software. The standard is described in [FIP01]. Brian Gladman wrote a detailed description of AES in [Gla01] and an updated version of the description in [Gla07]. We will give here a short overview to introduce the notation used in this thesis.

AES is a typical *substitution-permutation network* (SPN), see [Sti05]. An SPN consists of several rounds, each made up with a substitution and a permutation stage. The substitution stage obscures the relationship between elements of the plaintext and elements of the ciphertext, while the permutation stage makes of the influence of plaintext elements spread over the ciphertext.

For AES, the number of rounds depends on the key size. We focus on a key size of 128 bits, which corresponds to ten rounds. The 16 byte plaintext $P = (p_0, p_1, \ldots, p_{15})$ is internally arranged in a 4×4 matrix, the so-called *State matrix* $\mathbf{S}$. Every element $s_{r,c}$ of the State matrix (r denotes the index of the row and c the index of the column) equals to one byte. In Figure 2.1 we can see the individual transformations which are applied to the State matrix. Figure 2.2 shows how the plaintext P is initially mapped to the State matrix $\mathbf{S}$ and then to the ciphertext output, $C = (c_0, c_1, \ldots, c_{15})$.

In each round, four transformations are applied to the State matrix $\mathbf{S}$:

1. *Byte Substitution (SubBytes)*, which is a non-linear byte substitution operating independently on each byte ($s_{r,c}$) of the State matrix, using a substitution table called S-box. This S-box can be constructed from two transformations:
   a) an inversion in the finite field $GF(2^8)$, where the zero is mapped to itself; and
   b) a bitwise affine transformation.

2. *ShiftRows*, where the bytes $s_{r,c}$ in the State matrix are cyclically shifted. The first row is not rotated. The second row is rotated to the left by one position, row three is rotated to the left by two positions and row four is rotated to the left by three positions.

3. *MixColumns*, where every column of the State matrix is treated as a four-term polynomial and gets multiplied by a fixed 4×4 matrix over $GF(2^8)$ which is defined in [FIP01].

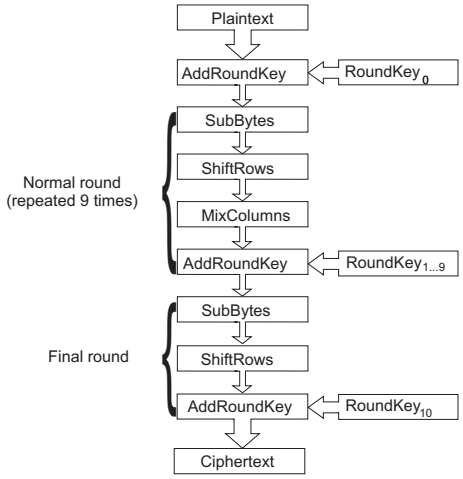

Figure 2.1: Sequence of transformations used in an AES-128 encryption

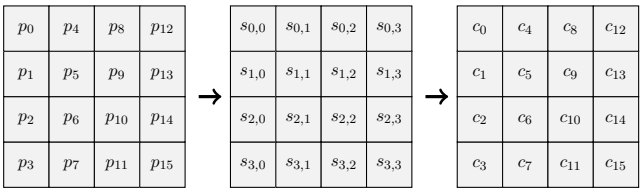

Figure 2.2: Mapping of input bytes to the State matrix and the output ciphertext

4. *Key Addition (AddRoundKey)*, which adds a 4×4 key matrix (128-bit) to the State matrix. This key matrix is called round key.

Figures 2.3 to 2.5 illustrate the transformations. The final round differs slightly from the first nine rounds. As we see in Figure 2.1 the *MixColumns* transformation is missing, which is also typical for a substitution-permutation network since the *MixColumns* transformation adds no cryptographic strength to the algorithm if it is included at the end of the cipher.

Decryption is accomplished by running the algorithm "backward". The last round key is first XORed with the ciphertext, and then in each round, the inverse linear transformations are applied, namely *InvMixColumns*, *InvShiftRows*, followed by the inverse S-box *InvSubBytes*. Note that for AES additional space for the inverse functions used in decryption is needed, in contrast to DES where encryption and decryption basically consume the same program memory. For example, in AES we need additional 256 bytes if the inverse transformation *InvSubBytes* is implemented as *lookup table* (LUT).

To supply the algorithm with the required round keys, AES uses a mechanism called key scheduling. The encryption key of 16 bytes is expanded to an array with eleven round keys, where each round key has a size of 16 bytes. So in total 176 bytes are needed for the round keys. To provide non-linearity between the cipher key and the expanded round key, the key schedule uses the *SubBytes* transformation followed by a cyclic rotation of the bytes and an addition of a round constant. The round constant is defined by a simple recursive rule.

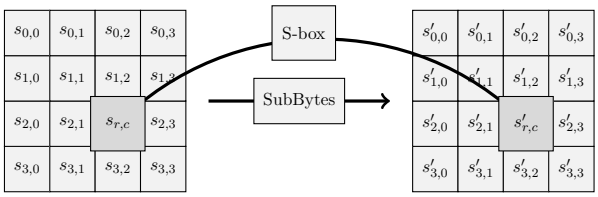

Figure 2.3: The SubBytes transformation

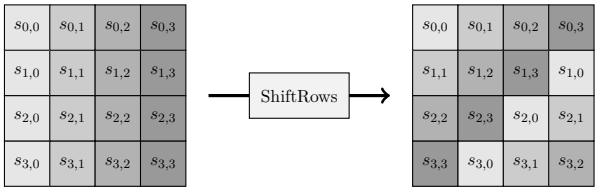

Figure 2.4: The ShiftRows transformation

2.2 Mathematical Background

In order to explain the upcoming implementation tricks and the side-channel countermeasures, we have to recall some basics on finite fields. Almost all operations that will follow are in the finite field GF(2), i. e., the bits '0' and '1', or the extension field GF(2^8), where GF denotes a Galois Field. A more detailed and general introduction to finite fields can be found in [Kna06].

GF(2^8) is the unique finite field with 256 elements and the operations $\oplus$ (addition) and $\otimes$ (multiplication). Its elements can be represented as polynomials of order 8 with binary coefficients $a_i \in \{0,1\}, i = 0,\ldots,7$. An element $a \in \mathrm{GF}(2^8)$ can thus be seen as the set of coefficients $a_i \in \mathrm{GF}(2)$ of the polynomial a with function value $a(x)$, i. e.,

$$\{a_7a_6a_5a_4a_3a_2a_1a_0\}_2 \mapsto a(x), \tag{2.1}$$

$$a(x) = \sum_{i=0}^{7} a_i x^i = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0. \tag{2.2}$$

A sequence of 8 bits, for example $\{11010100\}_2$, is called a byte. We will also depict byte values by their hexadecimal notation. Hence we can write the value $\{11010100\}_2$ as $\{\texttt{D4}\}_{16}$ which represents the polynomial $x^7 + x^6 + x^4 + x^2$.

Addition. In the following, $\oplus$ denotes an addition in GF(2^8), that can be realized using an exclusive OR (XOR). We can add two elements $a, b \in GF(2^8)$ by adding their coefficients modulo 2 element wise.

$$\sum_{i=0}^{7} a_i x^i \oplus \sum_{i=0}^{7} b_i x^i = \sum_{i=0}^{7} (a_i \oplus b_i) x^i \tag{2.3}$$

Multiplication. Multiplication in GF(2^8) is the usual polynomial multiplication, modulus an irreducible polynomial. For AES, the reduction polynomial m with

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{2.4}$$

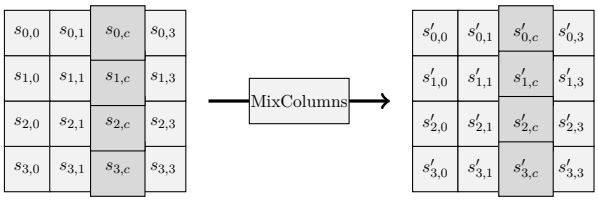

Figure 2.5: The MixColumns transformation

is used. The polynomial m can be represented by $\{\texttt{11B}\}_{16}$. Note that in $\mathrm{GF}(2^8)$ reduction by $m(x)$ is equivalent to adding $m(x)$.

We will use $\otimes$ to indicate the multiplication of two $\mathrm{GF}(2^8)$ elements, i. e., $a(x) \otimes b(x) := a(x) \cdot b(x) \bmod m(x)$ with $a, b \in \mathrm{GF}(2^8)$ and "$\cdot$" denoting a polynomial multiplication. The reduction by $m(x)$ ensures that the result is an element of $\mathrm{GF}(2^8)$.

We can implement the multiplication by x by a left shift and a subsequent conditional bitwise XOR with $\{\texttt{11B}\}_{16}$, if the most significant bit (MSB) bit is set. For example, multiplying $\{11001000\}_2$ by x gives $\{110010000\}_2$. The *overflow* bit is then removed by adding $\{100011011\}_2$, the modular polynomial, to give the final result of $\{10001011\}_2$.

This method is called *xtimes*, see [Gla07]. Multiplication by higher powers of x can be implemented by repeated application of xtimes. Since we can write all elements of $\mathrm{GF}(2^8)$ as a sum of powers of $\{\texttt{02}\}_{16}$, we can implement the multiplication by any value by a repeated use of xtimes. To get the result of the multiplication with any input g by the constant value $\{\texttt{15}\}_{16}$, for example, we compute:

$$\begin{aligned} g \otimes \{\texttt{15}\}_{16} &= g \otimes (\{\texttt{01}\}_{16} \oplus \{\texttt{04}\}_{16} \oplus \{\texttt{10}\}_{16}) \\ &= g \otimes (\{\texttt{01}\}_{16} \oplus \{\texttt{02}\}_{16}^2 \oplus \{\texttt{02}\}_{16}^4) \\ &= g \oplus \mathrm{xtimes}(\mathrm{xtimes}(g)) \oplus \mathrm{xtimes}(\mathrm{xtimes}(\mathrm{xtimes}(\mathrm{xtimes}(g)))) \\ &= g \oplus \mathrm{xtimes}(\mathrm{xtimes}(g \oplus \mathrm{xtimes}(\mathrm{xtimes}(g)))). \end{aligned}$$

2.3 Implementation Aspects

AES is an industry-wide used block cipher since the National Institute of Standards and Technologies (NIST) adopted it in 2001 from the Rijndael algorithm. It is used for the protection of government documents as well as for electronic commerce and private use. Many different implementations of AES have been proposed and have been optimized for the specific requirements like maximized throughput or minimized circuitry. We will give a short overview of the implementations, in order to get a picture of the tricks we can use for our implementation.

We begin by defining two microcontroller models (8-bit and 32-bit) to estimate the clock cycles for the implementation. Then we present the 8-bit implementation which is proposed in the standard [FIP01] and then look at existing 32-bit software implementations. We introduce the affine transformation, which is part of the *SubBytes* transformation, because we will need it in a later chapter.

For the current master thesis it is required to implement both encryption and decryption. For reasons of brevity, we will focus on the description of the encryption and give only the necessary hints for understanding the decryption process.

2.3.1 Microcontroller Model

To compare the runtime of an encryption, we define a simple 8-bit processor model and count the clock cycles (cc) according to that model. This model does not fit to all existing 8-bit microcontrollers. Hence, the given clock cycle count is a rough estimate to get an impression of the implementation. We make the following assumptions for byte operations on our processor model:

- load/store from/to RAM takes one clock cycle,
- load/store from/to ROM takes two clock cycles,
- XOR two registers takes one clock cycle,
- shift by one bit takes one clock cycle.

For the 32-bit (word) operations, we expand our processor model, which is defined in Section 2.3.3. We assume, that a load and a store operation on a word takes one clock cycle.

2.3.2 The Key Schedule

The AES implementation has a key size of 128 bits, which leads to ten rounds. With the initial *AddRoundKey* transformation, an entire encryption needs eleven round keys. There are two different options to supply the algorithm with those round keys:

Key Expansion Before the Encryption

```
1  KeyExpansion(byte key[16], byte expkey[4][44])
2    begin
3      for (j = 0; j < 4; j++) {
4        for (i = 0; i < 4; i++)
5          expkey[i][j] = key[j * 4 + i];
6      }
7      for (j = 4; j < 44; j++) {
8        if (j % 4 == 0) {
9          expkey[0][j] = expkey[0][j-4] ^
10                        SubBytes(expkey[1][j-1], 0) ^ rc[j/4];
11         for (i = 1; i < 4; i++)
12           expkey[i][j] = expkey[i][j-4] ^
13                        SubBytes(expkey[(i+1) % 4][j-1],0);
14       }
15       else {
16         for (i = 0; i < 4; i++)
17           expkey[i][j] = expkey[i][j - 4] ^ expkey[i][j - 1];
18       }
19     }
20   end
```

Listing 2.1: Key expansion from [DR02]

On the one hand, the whole round keys can be calculated before the actual encryption. Therefore, the *KeyExpansion* function is used, which is depicted in Listing 2.1. This function expands the 128-bit cipher key into an expanded key array, with four rows and 44 columns. The size of each element of this array equals to one byte. To store this expanded key array, we need $11 \cdot 16$ bytes $= 176$ bytes RAM. The first four columns (16 bytes) of the expanded key are filled with the 128-bit cipher key. The next columns recursively depend on the previous ones, with the following cases: If the column index $j = 0, \ldots, 43$ is not a multiple of four, column j is a bitwise XOR of column $j-4$ and column $j-1$. Otherwise, column j is a bitwise XOR of column $j-4$ and $j-1$, with the difference, that the S-box is applied on the four bytes of column $j-1$, with an additional XOR of the round constant rc on the first byte of this column. Since there are eleven rounds, also eleven round constants are needed, which are calculated by the following recursion rule in GF(2^8):

$$\text{rc}[1] = 1, \quad \text{rc}[2] = x, \quad \text{rc[n]} = x \otimes \text{rc}[n-1], \quad n = 3, \ldots, 11. \tag{2.5}$$

Key On-The-Fly Calculation

On the other hand, we can calculate the round keys on-the-fly. This alternative method is for environments with limited memory, e.g., smartcards. The calculation works the same way as we described above, but we only store the last calculated round key since we see that the calculations depend only on the columns $j-4$ to $j-1$, so we can derive the next one from it. In Listing 2.2 the function *NextRoundKey* calculates the next round key from the last used round key and the last used round constant *rc*. The first round key is the cipher key, the first round constant is one.

```
1  NextRoundKey(byte round_key[16], byte rc)
2    begin
3      round_key[0] = round_key[0] ^ SubBytes[round_key[13]] ^ rc;
4      round_key[1] = round_key[1] ^ SubBytes[round_key[14]];
5      round_key[2] = round_key[2] ^ SubBytes[round_key[15]];
6      round_key[3] = round_key[3] ^ SubBytes[round_key[12]];
7
8      for (i=4; i<16;i++) {
9        round_key[i] = round_key[i] ^ round_key[i-4];
10     }
11     rc = xtime(rc);          /* compute new round constant */
12   end
```

Listing 2.2: Key on-the-fly generation [Gla01]

If the key is calculated on-the-fly, we have to add the calculation costs to our encryption costs, since the calculation is done during every encryption. By counting the operations from Listing 2.2, 17 XORs, $3 \cdot 16 = 48$ load and store operations plus four load operations from ROM (for the S-box lookup) and two shift and an additional addition (XOR) for the *xtime* operation are necessary to calculate the next round key. Summarizing, we get 72 cc for calculating the next round key and in total, the key on-the-fly method costs $10 \cdot 72$ cc $= 720$ cc.

2.3.3 8-bit Software Implementation Straight from the Standard

On an 8-bit processor, AES can be programmed by simply implementing the different component transformations which are described in Section 2.1.

The AddRoundKey Transformation

During the *AddRoundKey* transformation, every byte of the State matrix **S** is XORed with one byte from the expanded key. For one byte $s_{r,c}$ of the State this takes:

- 1 cc to load the byte $s_{r,c}$ from RAM,
- 1 cc to load one byte of the expanded key from RAM,
- 1 cc to XOR both, and
- 1 cc to store the result.

To process one State matrix **S**, $16 \cdot 4\,\text{cc} = 64\,\text{cc}$ are needed. Since eleven round keys are added during an AES run, a whole AES en-/decryption needs $11 \cdot 64\,\text{cc} = 704\,\text{cc}$ for the *AddRoundKey* transformations.

The ShiftRows Transformation

For *ShiftRows* the implementation is straightforward. Here the bytes in the last three rows of the State matrix are cyclically shifted. For this operation, the State value is loaded from RAM and must be written back at its new position. We assume that these two operations take two clock cycles per byte of the State matrix. Then it takes $4 \cdot 2\,\text{cc} = 8\,\text{cc}$ per row and $3 \cdot 8\,\text{cc} = 24\,\text{cc}$ per transformation. For a whole AES en-/decryption with ten rounds we need $10 \cdot 24\,\text{cc} = 240\,\text{cc}$ for the ten *ShiftRows* transformations.

The SubBytes Transformation

The *SubBytes* transformation is implemented in the standard as a static lookup table (the S-box) of 256 bytes. For decryption, the inverse *SubBytes* transformation is needed, which results in a second 256-byte lookup table. Both tables are generally located in read-only memory (ROM). The transformation loads one byte from ROM, one byte from the State matrix, XORs them, and writes the result back to RAM. We estimate that these operations take five clock cycles. One *SubBytes* transformation takes $16 \cdot 5\,\text{cc} = 80\,\text{cc}$. For a whole AES en-/decryption with ten rounds this makes $10 \cdot 80\,\text{cc} = 800\,\text{cc}$ for the ten *SubBytes* transformations.

The MixColumns Transformation

The *MixColumns* transformation considers every column s_c ($0 \leq c \leq 3$) of the State matrix as a *four-term polynomial* over $GF(2^8)$. This column s_c is multiplied by a fixed polynomial a with:

$$a(x) = \{\texttt{03}\}_{16} \cdot x^3 + \{\texttt{01}\}_{16} \cdot x^2 + \{\texttt{01}\}_{16} \cdot x + \{\texttt{02}\}_{16} \tag{2.6}$$

and then reduced by a fixed polynomial l with $l(x) = x^4 + 1$. The reduction by l decreases the result so it fits to the column again. Since a is coprime to l it is invertible and all elements have an inverse. The multiplication can be rewritten with the fixed polynomial a:

$$s'_c(x) = a(x) \cdot s_c(x) \bmod l(x), \text{ for } 0 \leq c \leq 3 \tag{2.7}$$

as the matrix:

$$s'_{0,c} = \{02\}_{16} \otimes s_{0,c} \oplus \{03\}_{16} \otimes s_{1,c} \oplus \{01\}_{16} \otimes s_{2,c} \oplus \{01\}_{16} \otimes s_{3,c}, \tag{2.8a}$$
$$s'_{1,c} = \{01\}_{16} \otimes s_{0,c} \oplus \{02\}_{16} \otimes s_{1,c} \oplus \{03\}_{16} \otimes s_{2,c} \oplus \{01\}_{16} \otimes s_{3,c}, \tag{2.8b}$$
$$s'_{2,c} = \{01\}_{16} \otimes s_{0,c} \oplus \{01\}_{16} \otimes s_{1,c} \oplus \{02\}_{16} \otimes s_{2,c} \oplus \{03\}_{16} \otimes s_{3,c}, \tag{2.8c}$$
$$s'_{3,c} = \{03\}_{16} \otimes s_{0,c} \oplus \{01\}_{16} \otimes s_{1,c} \oplus \{01\}_{16} \otimes s_{2,c} \oplus \{02\}_{16} \otimes s_{3,c}. \tag{2.8d}$$

The indices $0, \ldots, 3$ indicate the byte in the column c of the current State $\mathbf{S}$. A multiplication with $\{02\}_{16}$ can be efficiently implemented with the *xtime* operation which is denoted in Listing 2.3[1] and described in Section 2.2.

```
1 byte xtimes(byte x)
2   begin
3     return (x<<1) ^ (((x>>7) & 1) * 0x1b)
4   end
```

Listing 2.3: The xtimes operation

We estimate five clock cycles for the `xtimes` operation. A multiplication with $\{03\}_{16}$ can be implemented as a multiplication with $\{02\}_{16}$ plus an additional XOR operation with the operand. So for computing equations (2.8a) to (2.8d) we need:

- $4 \cdot 1\,\text{cc} = 4\,\text{cc}$ for loading from RAM,
- $4 \cdot 1\,\text{cc} = 4\,\text{cc}$ for writing to RAM,
- $4 \cdot 3\,\text{cc} = 12\,\text{cc}$ for the additions (XOR),
- $4 \cdot 5\,\text{cc} = 20\,\text{cc}$ for the four fixed multiplications with $\{02\}_{16}$,
- $4 \cdot 6\,\text{cc} = 24\,\text{cc}$ for the four fixed multiplications with $\{03\}_{16}$.

This makes 64 cc per column, $4 \cdot 64\,\text{cc} = 256\,\text{cc}$ per round and $9 \cdot 256\,\text{cc} = 2304\,\text{cc}$ for all nine *MixColumns* transformations during an AES encryption.

The Inverse MixColumns Transformation

The *InvMixColumns* transformation is similar to *MixColumns* transformation, but it uses the inverse of a:

$$a^{-1}(x) = \{0B\}_{16} \cdot x^3 + \{0D\}_{16} \cdot x^2 + \{09\}_{16} \cdot x + \{0E\}_{16}. \tag{2.9}$$

1 Please note that implementing *xtime* this way leaks timing information, due to the conditional multiplication with $\{1B\}_{16}$. This will be discussed in Section 3.1.

The Hamming weight[2] of the coefficients of $a^{-1}(x)$ is larger, which leads to more `xtime` calls and thus to a slower implementation for the decryption. For the *InvMixColumns* operation, which is given by the equations:

$$s'_c(x) = a^{-1}(x) \cdot s_c(x) \bmod l(x), \text{ for } 0 \leq c \leq 3 \tag{2.10}$$

we get the following matrix:

$$s'_{0,c} = \{\texttt{0E}\}_{16} \otimes s_{0,c} \oplus \{\texttt{0B}\}_{16} \otimes s_{1,c} \oplus \{\texttt{0D}\}_{16} \otimes s_{2,c} \oplus \{\texttt{09}\}_{16} \otimes s_{3,c}, \tag{2.11a}$$
$$s'_{1,c} = \{\texttt{09}\}_{16} \otimes s_{0,c} \oplus \{\texttt{0E}\}_{16} \otimes s_{1,c} \oplus \{\texttt{0B}\}_{16} \otimes s_{2,c} \oplus \{\texttt{0D}\}_{16} \otimes s_{3,c}, \tag{2.11b}$$
$$s'_{2,c} = \{\texttt{0D}\}_{16} \otimes s_{0,c} \oplus \{\texttt{09}\}_{16} \otimes s_{1,c} \oplus \{\texttt{0E}\}_{16} \otimes s_{2,c} \oplus \{\texttt{0B}\}_{16} \otimes s_{3,c}, \tag{2.11c}$$
$$s'_{3,c} = \{\texttt{0B}\}_{16} \otimes s_{0,c} \oplus \{\texttt{0D}\}_{16} \otimes s_{1,c} \oplus \{\texttt{09}\}_{16} \otimes s_{2,c} \oplus \{\texttt{0E}\}_{16} \otimes s_{3,c}. \tag{2.11d}$$

For the inverse *MixColumns* transformation the following operations are needed:

- $4 \cdot 1\,\text{cc} = 4\,\text{cc}$ for loading from RAM,
- $4 \cdot 1\,\text{cc} = 4\,\text{cc}$ for writing to RAM,
- $4 \cdot 3\,\text{cc} = 12\,\text{cc}$ for the additions (XOR),
- $4 \cdot 17\,\text{cc} = 68\,\text{cc}$ for the four fixed multiplications with $\{\texttt{0E}\}_{16}$,
- $4 \cdot 17\,\text{cc} = 68\,\text{cc}$ for the four fixed multiplications with $\{\texttt{0D}\}_{16}$,
- $4 \cdot 17\,\text{cc} = 68\,\text{cc}$ for the four fixed multiplications with $\{\texttt{0B}\}_{16}$,
- $4 \cdot 16\,\text{cc} = 64\,\text{cc}$ for the four fixed multiplications with $\{\texttt{09}\}_{16}$,

Summarizing: To calculate a column of the State matrix $\mathbf{S}$, 288 cc are needed. To calculate the whole State matrix $4{\cdot}288\,\text{cc} = 1152\,\text{cc}$ and $9{\cdot}1152\,\text{cc} = 10368\,\text{cc}$ for all nine *InvMixColumns* transformations, during an AES decryption.

The Affine Transformation

As described in Section 2.1, the *SubBytes* transformation can be realized by a inversion followed by an affine transformation. Figure 2.6 illustrates these two steps. In the first part of the transformation the multiplicative inversion $s_{i,j}^{-1}$ in GF(2^8) is build of the processed byte $s_{i,j}$. The inversion is followed by an affine transformation. The affine transformation

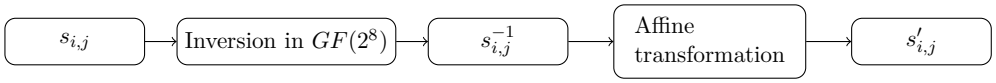

Figure 2.6: Decomposed SubBytes transformation

is a polynomial multiplication of a constant matrix with the byte $a = s^{-1}$ followed by a

2 No. of "1" in the binary representation

XOR with the constant c. The matrix and the constant c are given in the standard [FIP01]. Equation (2.12) denotes the affine transformation.

$$\begin{pmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \tag{2.12}$$

Because some of the countermeasures we will present calculate the inversion on the fly, we also have to calculate the affine transformation. So we estimate the runtime for this transformation. To calculate the runtime, we use the hypothetical processor model from Section 2.3.1 and count the clock cycles (cc). To calculate one bit of the result byte b we need 8 multiplications (AND) and seven additions (XOR). This makes $8+7 = 15$ cc for one result bit and $8 \cdot 15\,\text{cc} = 120\,\text{cc}$ for all eight resulting bits a_0 to a_7. The affine transformation is executed 160 times during a whole AES run. So this naive way would take us $160 \cdot 120\,\text{cc} = 19200\,\text{cc}$ just for the affine transformation.

Obviously, we can improve this implementation. If we take a closer look at the constant matrix in (2.12) the first thing we see, is the pattern $m = \{11111000\}_2$ in the first row. This pattern repeats in the following rows, but every time rotated by one position to the right. With the fact, that this calculation in done in GF(2), we can also save some computation. If we are able to calculate the parity[3] of a byte, we can safe the seven bitwise binary additions. Therefore we multiply the current column m of the constant matrix with the byte a and just set the output bit in b if the parity of the result is even. Algorithm 1 denotes this method.

Algorithm 1: Calculation of the affine transformation

Input: a
Output: b=aff_trans(a)

```
1  b = 0;
2  m = {11111000}_2;
3  c = {01100011}_2;
4  for i = 0 to 7 do
5      t = a ⊗ m;
6      b = rotate_bit_left(b);
7      b = b ⊕ parity(t);
8      m = rotate_bit_right(m);
9  end
10 b = b ⊕ c;
```

Now we one multiplication (AND) 1 cc, two rotations 2 cc, one parity calculation 1 cc and one addition (XOR) 1 cc to calculate one bit of the result. For all eight bits we need $8 \cdot 5\,\text{cc} =$

3 States whether the sum of all bits in the byte are even or odd.

40 cc. In the last step, the constant c is added to the result what also takes one clock cycle. To calculate the affine transformation for one byte this takes now 40 cc + 1 cc = 41*cc*. The affine transformation part of a whole AES run, now takes $160 \cdot 41$ cc = 6560 cc.

Summary

Summarizing, in our processor model an AES encryption needs 240 cc for the *ShiftRows* transformation, 800 cc for the *SubBytes* transformation, and 2304 cc for the *MixColumns* transformation. Depending on the key calculation method, this gives either a total cycle count of 4048 cc if separate key scheduling is used, or 4064 cc for the key on-the-fly method for an entire AES encryption.

The AES decryption needs a bit more clock cycles, because the *InvMixColumns* is slower than the *MixColumns* transformation. With the 10368 cc for an *InvMixColumns* transformation, the decryption takes 704 cc + 240 cc + 800 cc + 10368 cc = 12112 cc for an AES decryption with separate key scheduling.

2.3.4 8-bit Optimized Software Implementation

The discussed AES implementation can be improved by combining the *ShiftRows* and *SubBytes* transformation, see [Den07]. In addition, the inverse *MixColumns* transformation needed for the decryption can be rewritten in a clock cycle saving way. This leads to an optimized implementation as discussed in this section.

ShiftRows and SubBytes Combined

Since *ShiftRows* is only a permutation of the indices of the State matrix elements, it can be combined with the *SubBytes* transformation. With this combination, we safe the 240 cc of the *ShiftRows* transformation.

Improved inverse MixColumns

In [DR02], P. Barreto observes the following relation between the *MixColumns* polynomial $a(x)$, which has been defined in Equation (2.6), and the *InvMixColumns* polynomial $a^{-1}(x)$:

$$a^{-1}(x) = (\{\texttt{04}\}_{16} \cdot x^2 + \{\texttt{05}\}_{16}) \cdot a(x) \bmod (x^4 + \{\texttt{01}\}_{16}) \tag{2.13}$$

There are two important points. The Hamming weight of the coefficients is much smaller than in the original Equation (2.9) and it has only half as many coefficients. In matrix notation, the right part of Equation (2.13) becomes:

$$\begin{pmatrix} \{\texttt{02}\}_{16} & \{\texttt{03}\}_{16} & \{\texttt{01}\}_{16} & \{\texttt{01}\}_{16} \\ \{\texttt{01}\}_{16} & \{\texttt{02}\}_{16} & \{\texttt{03}\}_{16} & \{\texttt{01}\}_{16} \\ \{\texttt{01}\}_{16} & \{\texttt{01}\}_{16} & \{\texttt{02}\}_{16} & \{\texttt{03}\}_{16} \\ \{\texttt{03}\}_{16} & \{\texttt{01}\}_{16} & \{\texttt{01}\}_{16} & \{\texttt{02}\}_{16} \end{pmatrix} \cdot \begin{pmatrix} \{\texttt{05}\}_{16} & \{\texttt{00}\}_{16} & \{\texttt{04}\}_{16} & \{\texttt{00}\}_{16} \\ \{\texttt{00}\}_{16} & \{\texttt{05}\}_{16} & \{\texttt{00}\}_{16} & \{\texttt{04}\}_{16} \\ \{\texttt{04}\}_{16} & \{\texttt{00}\}_{16} & \{\texttt{05}\}_{16} & \{\texttt{00}\}_{16} \\ \{\texttt{00}\}_{16} & \{\texttt{04}\}_{16} & \{\texttt{00}\}_{16} & \{\texttt{05}\}_{16} \end{pmatrix} = \begin{pmatrix} \{\texttt{0E}\}_{16} & \{\texttt{0B}\}_{16} & \{\texttt{0D}\}_{16} & \{\texttt{09}\}_{16} \\ \{\texttt{09}\}_{16} & \{\texttt{0E}\}_{16} & \{\texttt{0B}\}_{16} & \{\texttt{0D}\}_{16} \\ \{\texttt{0D}\}_{16} & \{\texttt{09}\}_{16} & \{\texttt{0E}\}_{16} & \{\texttt{0B}\}_{16} \\ \{\texttt{0B}\}_{16} & \{\texttt{0D}\}_{16} & \{\texttt{09}\}_{16} & \{\texttt{0E}\}_{16} \end{pmatrix} \tag{2.14}$$

So the normal *MixColumns* transformation can be applied after a little precalculation of the State matrix. For the constant multiplication of $\{04\}_{16}$ two xtime calls and for the multiplication by $\{05\}_{16}$ two xtime calls plus one addition (XOR) are needed. Listing 2.4 denotes the preprocessing step. The word w (four bytes) is the current column of the State matrix.

```
1  word ImvprovedInvMixColumns(byte w[4])
2    begin
3      u = xtime(xtime(w[0] ^ w[2]));
4      v = xtime(xtime(w[1] ^ w[3]));
5
6      w[0] = w[0] ^ u;
7      w[1] = w[1] ^ v;
8      w[2] = w[2] ^ u;
9      w[3] = w[3] ^ v;
10
11     MixColumn(w);
12     return w;
13   end
```

Listing 2.4: Preprocessing step for improved InvMixColumns

The precalculation needs four *xtimes* operations and six XOR operations, these are $4 \cdot 5\,\text{cc} + 6 \cdot 1\,\text{cc} = 26\,\text{cc}$. The precalculation is followed by a normal *MixColumns* transformation which leads to $26\,\text{cc} + 64\,\text{cc} = 90\,\text{cc}$ to calculate the inverse *MixColumns* transformation for one column of the State matrix. A whole State **S** takes $4 \cdot 90\,\text{cc} = 360\,\text{cc}$ and all nine *MixColumns* transformations during an AES decryption take $9 \cdot 360\,\text{cc} = 3240\,\text{cc}$.

Summary

Summarizing, the optimized (8-bit) AES encryption takes 704 cc for the *AddRoundKey* transformation, 800 cc for the combined *ShiftRows* and *SubBytes* transformation, and 2304 cc for the *MixColumns* transformation. In total, this makes 3808 cc for one entire AES encryption with TLU keys.

With the improved inverse *MixColumns* transformation, the AES decryption now needs $704\,\text{cc} + 800\,\text{cc} + 3240\,\text{cc} = 4744\,\text{cc}$ to decrypt a 128 bit ciphertext.

2.3.5 32-bit Software Implementation

Two of the discussed 8-bit transformations, namely the *AddRoundKey* and the *MixColumns* transformation both benefit directly from a 32-bit register size. In the next sections, we discuss this 32-bit implementation. In addition, there is a 32-bit implementation called T-tables implementation, which combines the four transformations in precalculated tables.

The AddRoundKey Transformation

The *AddRoundKey* transformation can also be implemented to work simultaneously on a whole column of the State matrix. Therefore it loads one word from the expanded key array

and one word from the State matrix **S** and adds them. The resulting word is written back to the State matrix. To process the State **S**, $4 \cdot 4\,\text{cc} = 16\,\text{cc}$ are needed. All eleven *AddRoundKey* transformations now take $11 \cdot 16\,\text{cc} = 176\,\text{cc}$.

The MixColumns Transformation

The *MixColumns* transformation can be speed up considerably on 32-bit processors. Gladman mentioned in [Gla01] that equation (2.8a) to (2.8d) can be rewritten in such a way, that only multiplications by two are necessary, e. g., $\{03\}_{16} \otimes s_{0,c} = \{02\}_{16} \otimes s_{0,c} \oplus s_{0,c}$. We get the following equations:

$$s'_{0,c} = s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \oplus s_{0,c} \oplus \{02\}_{16} \otimes (s_{1,c} \oplus s_{0,c}), \tag{2.15a}$$
$$s'_{1,c} = s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \oplus s_{1,c} \oplus \{02\}_{16} \otimes (s_{2,c} \oplus s_{1,c}), \tag{2.15b}$$
$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \oplus s_{2,c} \oplus \{02\}_{16} \otimes (s_{3,c} \oplus s_{2,c}), \tag{2.15c}$$
$$s'_{3,c} = s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \oplus s_{3,c} \oplus \{02\}_{16} \otimes (s_{0,c} \oplus s_{3,c}), \tag{2.15d}$$

where the index c denotes the current column of the State matrix. These equations can be rewritten to:

$$s'_{0,c} = s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \oplus \{02\}_{16} \otimes (s_{1,c} \oplus s_{0,c}), \tag{2.16a}$$
$$s'_{2,c} = s_{2,c} \oplus s_{3,c} \oplus s_{0,c} \oplus \{02\}_{16} \otimes (s_{2,c} \oplus s_{1,c}), \tag{2.16b}$$
$$s'_{2,c} = s_{3,c} \oplus s_{0,c} \oplus s_{1,c} \oplus \{02\}_{16} \otimes (s_{3,c} \oplus s_{2,c}), \tag{2.16c}$$
$$s'_{3,c} = s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus \{02\}_{16} \otimes (s_{0,c} \oplus s_{3,c}). \tag{2.16d}$$

The four bytes $s_{0,c}$ to $s_{3,c}$ can be stored successively in the word w. Listing 2.5 denotes the multiplication by two within a whole word, called `FFmulX`. The word t, on the left hand side (from the second XOR) in line four, extracts the highest bits from each byte within the word w. The four individual bytes are now multiplied by two in parallel with a single left shift. On the right hand side, the bytes of word w whose top bits are set are multiplied by $\{1B\}_{16}$, the AES reduction polynomial. This reduction ensures, that the four 8-bit results fits into the 32-bit register.

```
1 word FFmulX(word w)
2   begin
3     word t = w & 0x80808080;
4     return ((w ^ t) << 1) ^ ((t >> 3) |
5                  (t >> 4) | (t >> 6) | (t >> 7));
6   end
```

Listing 2.5: Parallel multiplication of each byte in word w by two

Since the four bytes $s_{0,c}$ to $s_{3,c}$ are stored successively in the word w, the four Equations (2.16a) to Equations (2.16d) can be calculated in parallel. Listing 2.6 denotes this *MixColumns* transformation for one column of the State matrix. The resulting word contains the four transformed bytes $s'_{0,c}$ to $s'_{3,c}$.

```
1 word MixColumn(word w)
2   begin
3     w = rot1(w) ^ rot2(w) ^ rot3(w) ^ FFmulX(rot1(w) ^ w);
4     return w;
5   end
```

Listing 2.6: MixColumns transformation for one column of the State matrix

With Listing 2.5 and Listing 2.6 we can estimate the clock cycles for the *MixColumns* transformation. The `FFmulX` function needs ≈ 11 cc. The `MixColumn` operation needs four rotations, four XOR operations, one `FFmulX` operation which makes 19 cc. In addition it has to load the word w from RAM and store the result back in RAM, this takes two clock cycles. One *MixColumns* transformation needs four `MixColumn` operations which makes $4 \cdot 21\,\text{cc} = 84\,\text{cc}$ for one round and $9 \cdot 84\,\text{cc} = 756\,\text{cc}$ for all nine *MixColumns* transformations.

The Inverse MixColumns Transformation

The 8-bit precalculation from Listing 2.4 can be formulated in such a way, that it uses 32-bit operations. Listing 2.7 denotes the improved inverse *MixColumns* transformation for one column w of the State matrix.

```
1  word InvMixColumn(word w)
2    begin
3      word tmp;
4
5      tmp = rot2(w);
6      tmp = tmp ^ w;
7      tmp = FFmulX(tmp);
8      tmp = FFmulX(tmp);
9
10     w = w ^ tmp;
11
12     w = MixColumn(w);
13
14     return w;
15   end
```

Listing 2.7: Inverse MixColumns transformation with 32-bit operations

We can estimate the clock cycles for the *InvMixColumns* transformation with Listing 2.7. For one column of the State matrix it takes one rotation, two XOR operations, two `FFmulX` operations and one `MixColumn` operation. This makes $1\,\text{cc} + 2 \cdot 1\,\text{cc} + 2 \cdot 11\,\text{cc} + 21\,\text{cc} = 46\,\text{cc}$ for one column, $4 \cdot 46\,\text{cc} = 184\,\text{cc}$ for a State and $9 \cdot 184\,\text{cc} = 1656\,\text{cc}$ for all nine rounds.

Summary

Summarizing the optimized 32-bit AES encryption takes 176 cc for the *AddRoundKey* transformation, 800 cc for the combined *ShiftRows* and *SubBytes* transformation, and 756 cc for

the *MixColumns* transformation. In total this makes 1732 cc for the encryption.

The decryption, with 1656 cc for the *InvMixColumns* transformation, now needs 176 cc + 800 cc + 1656 cc = 2632 cc.

2.3.6 32-bit T-Tables Implementation

Daemon and Rijmen already showed in their original AES proposal, that AES can be implemented very efficiently on 32-bit processors by combining the transformations *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey* for one column[4]. They define four tables each of 256 4-byte words (for $0 \leq x \leq 255$) as follows:

$$T_3[x] = \begin{pmatrix} \{02\}_{16} \otimes \text{S-box(x)} \\ \{03\}_{16} \otimes \text{S-box(x)} \\ \text{S-box(x)} \\ \text{S-box(x)} \end{pmatrix}, \qquad T_2[x] = \begin{pmatrix} \text{S-box(x)} \\ \{02\}_{16} \otimes \text{S-box(x)} \\ \{03\}_{16} \otimes \text{S-box(x)} \\ \text{S-box(x)} \end{pmatrix}, \tag{2.17a}$$

$$T_1[x] = \begin{pmatrix} \text{S-box(x)} \\ \text{S-box(x)} \\ \{02\}_{16} \otimes \text{S-box(x)} \\ \{03\}_{16} \otimes \text{S-box(x)} \end{pmatrix}, \qquad T_0[x] = \begin{pmatrix} \{03\}_{16} \otimes \text{S-box(x)} \\ \text{S-box(x)} \\ \text{S-box(x)} \\ \{02\}_{16} \otimes \text{S-box(x)} \end{pmatrix}. \tag{2.17b}$$

An AES round can be built from the following four table lookups, one for every column in the State matrix:

$$\begin{pmatrix} s_{3,c} \\ s_{2,c} \\ s_{1,c} \\ s_{0,c} \end{pmatrix} = T_3[s_{3,c(3)}] \oplus T_2[s_{2,c(2)}] \oplus T_1[s_{1,c(1)}] \oplus T_0[s_{0,c(0)}] \oplus k_{\text{round} \cdot 4+c}, \tag{2.18}$$

where $c(r) = c + r \bmod 4$ with $c(0) = c$. Here c denotes the actual column index, and $c(r), r = 0, \ldots, 3$ denotes the new position after the *ShiftRows* transformation. With these tables each column in the output State s' can be computed by using four XOR operations together with one word from the key schedule and four table-lookups that are indexed using four bytes from the input State. We need four XOR and five TLU per column. Since we have four columns per State matrix, $4 \cdot 4 = 16$ XOR, $4 \cdot 4 = 16$ TLU (ROM) and four TLU (RAM) per round are needed. This makes $16\,\text{cc} + 2 \cdot 16\,\text{cc} + 4\,\text{cc} = 52\,\text{cc}$ for one round and $10 \cdot 52\,\text{cc} = 520\,\text{cc}$ for the whole algorithm (plus additional key scheduling).

One table consumes $4 \cdot 256$ bytes = 1024 bytes ROM, and for the four tables $4 \cdot 1024$ bytes = 4096 bytes are needed. In the last round, the *MixColumns* is not present. This means that different tables are required for the last round. If we also implement the last round as a table, we need $2 \cdot 4096$ bytes = 8192 bytes of table space altogether.

If we take a closer look at the tables, it can be recognized that the structure of the four tables is closely related to each other. The columns are rotated by one, two and three bytes, since $T_i(x) = \text{rot}(T_{i-1}(x)), i = 0, \ldots, 3$, where the function `rot()` rotates the bytes in the column cyclically by one position. So, the space needed for the tables can be reduced by a factor of four at the expense of three additional cyclically rotations per round.

4 As far as we know, the first implementation using this structure was written by Rijmen, Bosselaers, and Barreto according to [BS08].

2.3.7 32-bit Implementation with Transposed State

Bertoni *et al.* [BBF+03] had the idea to transpose the State matrix **S** in such a way, that it can be processed by columns instead of rows. Therefore they have restructured the *MixColumns*, the *ShiftRows*, and the round key generation for en-/decryption. Since the *SubBytes* transformation works on each byte individually, they left it as it is.

The consequence of the transposition is that the new version of the *InvMixColumns* achieves a much higher speed gain. In fact, the transposed *InvMixColumns* requires seven doublings and 27 additions. The authors report a cycle count of 2047 cc for decryption on an `ARM7TDMI`. The encryption becomes a bit slower than the 32-bit reference implementation of Gladman. For encryption they need 1675 cc. Their reference implementation from Brian Gladman needs 2763 cc for decryption and 1641 cc for encryption on the `ARM7TDMI`.

2.3.8 Hardware Implementations

There has also been much effort put in efficient hardware implementations of AES: From the 8-bit implementation straight from the standard to 32-bit implementations using a 32-bit data path up to high-speed implementations with a 128-bit data path. For every implementation, very much care has to be given to the S-box implementation, both for performance and for security reasons.

Rijmen suggested in 2001 to use subfield arithmetic for computing the inverse part of the *SubBytes* transformation in [Rij01]. This idea was extended by Satoh *et al.* in [SMTM01] by using sub-subfields, the so-called "composite field" or "tower field" approach, which resulted in the smallest AES circuit at that point. The "composite field" approach utilizes that a finite field $\mathrm{GF}(2^k)$ with $k = n \cdot m$ can also be represented as $\mathrm{GF}((2^n)^m)$. Later, Canright [CB09] improved this method again by a carefully chosen normal basis, which results in the most compact S-box up to date.

As we will see, some ideas from the "composite field" approach can be reused for our SCA resistant AES implementation. So we will give a short overview to gain insight into the "composite field" arithmetic.

Composite Fields

We have seen in Section 2.2, that the finite field $\mathrm{GF}(2^8)$ can be seen as an extension field of $\mathrm{GF}(2)$ and its elements are represented as bytes. However, we can also see $\mathrm{GF}(2^8)$ as a quadratic extension of the field $\mathrm{GF}(2^4)$, which is far better suited for hardware implementations [Paa94, SMTM01]. We call this field *composite field*. In particular, composite field inversions are used to create compact AES hardware implementations [RDJ+01, SMTM01, MS03, WOL02].

We can represent every element $a \in \mathrm{GF}(2^8)$ as a linear polynomial $a_h x + a_l$ with the coefficients a_h and a_l being elements of the field $\mathrm{GF}(2^4)$. We refer to this polynomial as *two-term polynomial*. Both coefficients of this polynomial have four bits. The finite field $\mathrm{GF}(2^8)$ is isomorphic to the field $\mathrm{GF}((2^4)^2)$. For each element in $\mathrm{GF}(2^8)$ there exists exactly one element in $\mathrm{GF}((2^4)^2)$. The following field polynomial, which is used to define the quadratic extension of $\mathrm{GF}(2^4)$, is taken from [WOL02, Tri03].

Two two-term polynomials can be added by adding their corresponding coefficients:

$$(a_h x + a_l) \oplus (b_h x + b_l) = (a_h \oplus b_h)x + (a_l \oplus b_l). \tag{2.19}$$

For the multiplication and inversion of two two-term polynomials a modular reduction step is required, to ensure that the result is again a two-term polynomial. The irreducible polynomial n used for the reduction is given by:

$$n(x) = x^2 + x + \{\mathtt{E}\}_{16}. \tag{2.20}$$

The coefficients of n are elements of $\mathrm{GF}(2^4)$ and written in hexadecimal notation. To ensure, that the product of two coefficients lies in $\mathrm{GF}(2^4)$, i. e., $a(x) \otimes b(x) := a(x) \cdot b(x) \bmod m_4(x)$ with $a, b \in \mathrm{GF}(2^4)$, the irreducible reduction polynomial m_4 is used, which is given by:

$$m_4(x) = x^2 + x + 1. \tag{2.21}$$

The inversion in $\mathrm{GF}((2^4)^2)$ can be computed by using only the operations which have been defined above in $\mathrm{GF}(2^4)$:

$$(a_h x \oplus a_l)^{-1} = a'_h x \oplus a'_l \tag{2.22a}$$
$$a'_h = a_h \otimes d' \tag{2.22b}$$
$$a'_l = (a_h \oplus a_l) \otimes d' \tag{2.22c}$$
$$d = ((a_h^2 \otimes \{\mathtt{E}\}_{16}) \oplus (a_h \otimes a_l) \oplus a_l^2) \tag{2.22d}$$
$$d' = d^{-1} \tag{2.22e}$$

To calculate the inverse of d Oswald *et al.* shift this element down to $GF(2^2)$ in [OMP04], where they compute the inversion on a dedicated hardware. This is feasible in hardware but it is not efficient in software on a microcontroller. Instead the inversion of d can be efficiently realized as a lookup table with 16 elements from $GF(2^4)$.

If the inversion part of *SubBytes* is calculated with the composite field method, the big lookup table for the inversion in $\mathrm{GF}(2^8)$ with 256 elements can be replaced by the lookup table for the inversion in $\mathrm{GF}(2^4)$ with 16 elements.

2.3.9 Comparison of Different Implementation Options

In the last sections, several options to implement AES were discussed. Two main possibilities to write an 8-bit software implementation of AES have been pointed out.

In the first method, all round keys are calculated in advance, Table 2.1 line one, three, four, five and six. A method with separate key scheduling can be recommended, if there is enough RAM to store the 176 bytes for the round keys. The separate key scheduling especially pays off, if two or more blocks of 128 bit are encrypted[5]. An AES 128-bit encryption with separate key scheduling takes 3808 cc on an 8-bit processor (line three) and 1732 cc on a 32-bit processor (line four).

The second method is to implement an embedded key scheduling. Here the round keys are calculated when they are needed, on-the-fly, and afterwards they are discarded. This leads

5 Because the pre-computation of the expanded key (the key scheduling) has only to be done once.

to a very small RAM consumption. Table 2.1 denotes in line two, that this implementation needs 4064 cc and only 16 bytes RAM (for the last used round key). If the processor used has a large number of registers, everything can be calculated without the use of RAM. Thus, the on-the-fly method can be preferred on architectures where only very limited RAM is available.

On a 32-bit processor, AES can also be implemented with almost only table lookups. Table 2.1 refers to this method as *T-tables*. An 128-bit encryption with the T-tables approach takes approximately 520 cc (with separate key scheduling). The T-tables implementation needs 8192 bytes of ROM, to store the four pre-calculated tables that are needed for the encryption, and additional 8192 bytes for the decryption.

The first three entries in the table are to get a rough feeling of the space and cycle consumption of AES. In the second part of the table they can be compared with real implementations. We notice that there are big clock cycles differences between different implementations even if they are on the same architecture (8 or 32-bit). So we just compare the ratio between the cycle counts for an implementation given from the same author.

The cAESar implementation from [HKQ99] is one of the first 8-bit implementations we found. The used processor is not clearly described in the paper. The implementation is completely written in Assembler and needs 2889 cc for one encryption. The round keys are generated on-the-fly.

The second implementation is from Akkar/Giraud in 2001 [AG01] (line 8). Their non-optimized implementation was done in assembly code using an 8-bit processor. In their paper, they specify the runtime with 18.1 ms at 5 MHz which results in 90500 cc for the encryption of a 128-bit message. The whole implementation needs 730 bytes ROM and 41 bytes of RAM.

In [HOM06], Herbst *et al.* compare their SCA resistant 8-bit implementation with one self made older unprotected one from the Institute for Applied Information Processing and Communication (IAIK) [IAI06] they work at. This unprotected implementation needs 4427 cc for an encryption. The second unprotected implementation they refer to, is an implementation from Christian Röpke [RÖ3] on an ATmega163, which needs 7498 cc for an encryption. These two clock cycle counts are interesting for us, because we want use to some countermeasures from Herbst *et al.*

Bernstein and Schwabe [BS08] have written the fastest 32-bit implementation we have found so far. For achieving these results, they have made heavily optimized versions for the underlying 32-bit micro architecture and utilized the T-tables approach. On an Intel Core 2 Quad with, e. g., combined load and XOR operations, their implementation needs ≈ 169 cc for the encryption. In their clock cycle calculation they ignored the costs for computing the round keys.

The main requirement for our AES implementation is to be small in terms of RAM and ROM consumption. For this reason, we do not take the 32-bit T-tables approach. Because a typical AES implementation will have a fixed key and we assume that more than one 128-bit block is encrypted during one AES run, we decided us for the separate key scheduling method. Therefore we will take the optimized 32-bit implementation (line four) and try to implement it on the TriCore architecture as fast as possible. The RAM consumption in Table 2.1 does not contain the 16-byte State matrix.

No.	Implementation	ROM (bytes)	RAM (bytes)	Encryption (clock cycles)	Encryption (clock cycles/byte)
Estimated Values					
1	8-bit w/separate key scheduling	256	176	4048	253
2	8-bit w/key on-the-fly	256	16	4064	288
3	8-bit optimized	256	176	3808	238
4	32-bit optimized	256	176	1732	108.25
5	32-bit optimized (dec)	256	176	2632	164.5
6	32-bit T-tables	8192	176	520	32.5
8-bit Software implementations					
7	cAESar [HKQ99]	256	0	2889	181
8	Akkar / Giraud [AG01]	730	41	90500	5656
9	Christian Röpke [RÖ3]	NA	NA	7498	469
10	IAIK [IAI06]	NA	NA	4427	277
32-bit Software implementations					
11	Transposed State [BBF+03]	256	0	1675	104.69
12	Bernstein / Schwabe [BS08]	NA	176	169	10.57
13	Tillich [THM07]	NA	176	1637	102

Table 2.1: Comparison between different unprotected AES software implementations

Chapter 3

Side-Channel Analysis

In the classical "cryptanalysis", a cryptographic algorithm (cipher) is an abstract mathematical object, which turns some plaintext input data into encrypted output data, parameterized by a key. Here, Kerckhoffs' law [Ker83] is a main rule for everybody. It means, that the security of a cryptosystem should not depend on the secrecy of the algorithm. All security is in the cryptographic key, which therefore needs to be carefully protected. In classical cryptanalysis the security of a cipher is investigated by mathematical proofs and analysis of the algorithm.

However, a cipher has to be implemented in a program, that will run on a given processor in a specific environment like a smartcard or embedded system. These systems have specific physical characteristics. If these physical characteristics deliver information about the secret key involved in the operation it is called a *side-channel*. Typical side-channels are:

- the time, an algorithm (or part of it) needs for execution,
- the power consumption of the device,
- the electromagnetic field of the device during its processing, and
- the specific behavior after injecting a fault.

The side-channel attacks attempt to recover the secret key by parts. By only looking at small parts of the key, exhaustive key search on these sub-keys becomes possible. This process is then repeated until the full key is found. In the following sections, we take a closer look at the first two side-channels and how they can be used to obtain information about the secret key.

3.1 Timing Analysis

If the runtime of a cryptographic device is key dependent, it can be used as a side-channel. Kocher introduced this idea in [Koc96]. A formalized description can be found in [HMS00], we will give here a short overview. For example, we want to discover the private exponent e in the following calculation:

$$y = x^e \bmod n. \tag{3.1}$$

We assume, that the calculation is done with the Square-and-Multiply method (left-to-right binary exponentiation) which is described in Algorithm 2. We see, that the running time of the algorithm directly depends on the bits $e = (e_p, \ldots, e_0)$ of the secret exponent where $p+1$ denotes the total number of bits in the exponent. If a bit of the exponent is one, then an

additional multiplication by the base x and reduction by n is performed, which will take a bit more time.

Algorithm 2: Left-to-right binary exponentiation (Square-and-Multiply)

Input: $x, n, e = (e_{p,\ldots,e_0})_2$
Output: $y = x^e \bmod n$

```
1 y = 1;
2 for i = p down to 0 do
3     y = y · y mod n;
4     if e_i = 1 then
5         y = y · x mod n;
6     end
7 end
```

Let us assume, that we already know the first bits $e_p, \ldots, e_{m-1}$ of the exponent. Bit e_m will be attacked. We measure the runtime $T_i, i = 1, \ldots, N$, for N modular exponentiations with random (but known) base x_i:

$$T_i = T(x_i^e \bmod n). \tag{3.2}$$

Next we assume that the unknown bit e_m is zero. We compute the execution time $T_{i,m}^0$ via an emulation of the runtime until bit e_m is reached. Now we build the empirical variance[1] $V_0 := \mathrm{Var}(T_i - T_{i,m}^0)$. We do the same for $e_m = 1$ and build the empirical variance $V_1 := \mathrm{Var}(T_i - T_{i,m}^1)$. Then we decide whether the exponent bit e_m was zero or one:

$$e_m = \begin{cases} 0 & \text{if } V_0 < V_1, \\ 1 & \text{else.} \end{cases} \tag{3.3}$$

This method works, because a wrong guess of e_m leads to an increase of the empirical variance, if we can exactly simulate the execution time. Note that we do not require new measurements for the other unknown bits. We just repeat the steps for the next unknown bit of the exponent e.

We see, that this attack is applicable, since intermediate values of the algorithm we attack depend on small parts of the key and lead to a different timing. This fact is utilized in other attacks, too. Since we know of the existence of those attacks, we will show in the following a method to verify, if our implementation is vulnerable to timing attacks.

General countermeasures against timing attacks are to modify the algorithm in a way, that it:

- always has a constant, thus key-independent, runtime,
- or masking the data, for example, with Chaum's blinding method presented in [Cha82]. Here the exponentiation is done with a value $\tilde{e}_i = \lambda e_i$, λ random, unknown to the attacker. Thus, the attacker cannot simulate the execution time $T_{i,m}^b$.

1 We define the empirical variance as: $v^2 = \frac{1}{N}\sum_{i=1}^{N}(t_i - \overline{t})^2$, where $\overline{t}$ is the mean $\frac{1}{N}\sum_i^N t_i$.

3.1.1 Evaluation of Timing Analysis Resistance

The Square-and-Multiply algorithm has demonstrated, that cryptographic algorithms can leak critical information during runtime. This example showed a very clear key-dependent runtime behavior. Often this behavior is not so clear, i. e., there could be even very small timing differences in the so-called Square-and-Always-Multiply Algorithm, a "fixed" version of Square-and-Multiply where a multiplication is performed regardless of the exponent bit. The "useless" result of this multiplication is then discarded. Obviously, in this case the timing difference between $e_m = 0$ and $e_m = 1$ is very small (if there is any). In order to detect such small variations we cannot rely on simple mechanisms, we need a thorough statistical background. Therefore, to verify if our AES implementation leaks key dependent information, some statistical tests are performed.

In the following, we assume that the running time of the algorithm is a random variable X which is distributed like a Gaussian distribution[2], i. e., $X \sim N(\mu_X, \sigma_X^2)$ with mean μ_X and variance σ_X^2.[3]

For the statistical analysis, we will perform runtime measurements with different plaintexts and different keys. We define some measurement sets with, for example, $N = 1000$ measurements per set. A set contains the plaintext, key and running time for an encryption run. Since the runtime varies and the measurement process is not perfect, there are always differences in the measured time.[4] The goal of the statistical tests, to be introduced in a minute, is to verify if these outliers or differences are systematic or not, i. e., to test if we can distinguish the measurement sets at hand of their runtime.

3.1.2 Two Sample T-Test

The density function of a *t-distributed* random variable X with $m \in \mathbb{N}$ degrees of freedom[5] is given by

$$f_{X,m}(x) = \frac{\Gamma(\frac{m+1}{2})}{\sqrt{\pi m}\,\Gamma(\frac{m}{2})}\left(1 + \frac{x^2}{m}\right)^{-\frac{m+1}{2}} \quad \text{for } x \in \mathbb{R}, \tag{3.4}$$

where Γ denotes the Gamma function, which is defined the following way

$$\Gamma(1) = 1, \quad \Gamma(n+1) = n! \quad \text{for } n \in \mathbb{N}$$

and can be generalized to arbitrary real arguments $x \in \mathbb{R}$. For sufficiently large m, the density function of the t-distribution approximates the Gaussian normal distribution $N(0,1)$, i. e., $f_{X,m} \xrightarrow{m\to\infty} N(0,1)$.

If X is an $N(0,1)$-normal distributed random variable and Y is a chi-squared (χ^2) distributed random variable with m degrees of freedom and both random variables are indepen-

2 For further details to statistics, e. g. probability density functions, statistical tests, etc., we refer to any standard statistics book, for example, [WG04] [Bos98].

3 More exactly, the running time of the algorithm for a deterministic input is composed of a deterministic part and a random part which is caused by algorithmic noise, cache and pipelining behavior etc. We assume that this random part is normally distributed.

4 For example, a cache could be flushed while we take a measurement set, which could result in a different running time for this specific measurement.

5 describes the number of values in the final calculation of a statistic that are free to vary

dent, then the random variable

$$T = \frac{X}{\sqrt{\frac{Y}{m}}}$$

is *t-distributed* with m degrees of freedom.

Now we can formulate the two sample t-test: Let $X_i, i = 1, \ldots, m$, and $Y_j, j = 1, \ldots, n$, be independent and identically distributed (i.i.d) random variables, respectively, with

$$X_i \sim N(\mu_X, \sigma_X^2) \qquad \text{and} \qquad Y_j \sim N(\mu_Y, \sigma_Y^2)$$

with equal but unknown variances $\sigma_X^2 = \sigma_Y^2$. Under the hypothesis $H_0 : \mu_X = \mu_Y$, the test characteristic

$$T = \frac{\overline{X} - \overline{Y}}{S_P} \sqrt{\frac{mn}{m+n}} \tag{3.5a}$$

with

$$S_P^2 = \frac{1}{m+n-2} \left((m-1)S_X^2 + (n-1)S_Y^2 \right), \tag{3.5b}$$

$$S_X^2 = \frac{1}{m-1} \sum_{i=1}^{m} (X_i - \overline{X})^2, \tag{3.5c}$$

$$S_Y^2 = \frac{1}{n-1} \sum_{j=1}^{n} (Y_j - \overline{Y})^2 \tag{3.5d}$$

has a t-distribution with $m + n - 2$ degrees of freedom.

To test, if both random variables come from the same population, we make the following two hypotheses:

$$H_0 : \mu_X = \mu_Y \qquad \text{and} \qquad H_1 : \mu_X \neq \mu_Y. \tag{3.6}$$

With a significance level α, for example, $\alpha = 0.05$, the hypothesis H_0 can be rejected, if

$$|T| \geq t_{m+n-2;1-\frac{\alpha}{2}}, \tag{3.7}$$

where $t_{m+n-2;1-\frac{\alpha}{2}}$ denotes the $1 - \alpha/2$ quantile of the t-distribution with $m + n - 2$ degrees of freedom.

Otherwise we have to say, that we have insufficient evidence to reject the null hypothesis.

3.1.3 Fisher's F-Test

With the t-test we test the mean values of our measurements under the assumption that the unknown variances are the same. To statistically test the observed variances, the so-called Fisher F-test can be used which will be described in this section. With the F-test we can verify statistical moments of order two, which are usually much smaller and more difficult to detect.

Let us assume two normally distributed random variables. To test if the variance of one random variable equals the variance of another random variable with unknown, but not necessarily equal mean values, we can use the F-test.

The density function of an *F-distributed* random variable X with (m, n)-degrees of freedom, $m, n \in \mathbb{N}$, is given by

$$f_X(x) = \begin{cases} \frac{\Gamma(\frac{m+n}{2}) m^{\frac{m}{2}} n^{\frac{n}{2}}}{\Gamma(\frac{m}{2})\Gamma(\frac{n}{2})} \cdot \frac{x^{\frac{m}{2}-1}}{(n+mx)^{\frac{m+n}{2}}} & \text{for } x > 0, \\ 0 & \text{for } x \leq 0. \end{cases} \tag{3.8}$$

If X and Y are two independent χ^2-distributed random variables with m and n degrees of freedom, $X \sim \chi^2_m, Y \sim \chi^2_n$, then the random variable

$$T = \frac{X/m}{Y/n}$$

is F-distributed with (m, n)-degrees of freedom.

Let $F_{m,n;\alpha}$ be the quantile with the order α. For large n the quantile of the F-distribution with (m, n)-degrees of freedom converges to the quantile of the χ^2-distribution with m-degrees of freedom multiplied with the factor $1/m$:

$$F_{m,n;\alpha} \xrightarrow{n\to\infty} \frac{\chi^2_{m;\alpha}}{m}.$$

Now we are able to formulate Fisher's F-test: Let $X_i, i = 1, \ldots, m$, and $Y_j, j = 1, \ldots, n$, be independent and identically distributed (i.i.d) random variables, respectively. We assume, that the random variables are normally distributed

$$X_i \sim N(\mu_X, \sigma^2_X) \qquad \text{and} \qquad Y_j \sim N(\mu_Y, \sigma^2_Y)$$

with μ_X and μ_Y unknown. Under the hypotheses $H_0 : \sigma^2_X = \sigma^2_Y$, the test statistic

$$T = \frac{S^2_X}{S^2_Y} \tag{3.9a}$$

with

$$S^2_X = \frac{1}{m-1} \sum_{i=1}^{m} (X_i - \overline{X})^2, \tag{3.9b}$$

$$S^2_Y = \frac{1}{n-1} \sum_{j=1}^{n} (Y_j - \overline{Y})^2 \tag{3.9c}$$

is F-distributed with $(m-1, n-1)$ degrees of freedom.

We make the following hypotheses

$$H_0 : \sigma^2_X = \sigma^2_Y \qquad \text{and} \qquad H_1 : \sigma^2_X \neq \sigma^2_Y. \tag{3.10}$$

With a significance level α, for example, $\alpha = 0.05$, we can reject the hypothesis H_0, if

$$|T| \geq F_{m-1,n-1;1-\frac{\alpha}{2}}. \tag{3.11}$$

Otherwise we have to say, that we have insufficient evidence to reject the null hypothesis.

3.2 Simple Power Analysis

At CRYPTO '99, Kocher *et al.* [KJJ99] introduced the so-called *power analysis attacks*. Every hardware device which performs cryptographic operations can leak information by its power consumption. A common example of an algorithm which might be vulnerable to a *Simple Power Analysis* (SPA) is the Square-and-Multiply algorithm which was discussed above. Because the algorithm goes sequentially through the bits of the exponent, the difference whether a multiplication takes place or not can be directly observed in the *power trace*. The power trace can be obtained by measuring the power consumption. This can be done by adding a small resistor in series with the power or ground of the attacked device. The current, which can be derived from the voltage across that resistor divided by the resistance, yields to the power trace. The length of the power trace is the product of the time span in seconds with the sampling rate. A simple power analysis is typically performed on one single power trace. However, multiple power traces can be recorded while the device performs the same operation on the same data several times. Computing the mean of those power traces will reduce the noise. Messerges has observed several sources for noise in [Mes00]. Sources for noise are, for example, at the clock edges, during the quantization and algorithm noise.

Building a good measurement setup should not be underrated in side-channel attacks. From an attackers viewpoint it is important, to avoid noise as much as possible. Also, if multiple traces for the same operation were recorded, they have to be aligned exactly in time. This can be achieved with an exact trigger and alignment techniques. Hermann Seuschek has analyzed the different types of noise in his master thesis [Seu05] and gives a detailed explanation on how to minimize these effects for an optimal measurement setup.

3.3 Differential and Correlation Power Analysis

The differential power analysis (DPA) exploits the relationship between the processed data and the power leakage of multiple traces with different input data. For a DPA, the detailed description of the implementation of the algorithm is not needed, which makes it a very powerful attack. It was suggested by Kocher [KJJ99] and later formalized by Messerges *et al.* in [MDS99]. The four following requirements, which are necessary for a first-order DPA attack on AES, can be formulated:

1. The attacker needs physical access to the device to measure the power consumption while it performs several en-/decryptions;

2. We need to know either the plaintext or the ciphertext of the algorithm (attack the first or the last rounds of the algorithm), we refer to it as the processed data vector $\mathbf{d} = (d_1, \ldots, d_i, \ldots, d_D)$, where d_i denotes the plaintext or ciphertext of the i^{th} decryption or encryption run and D is the total number of runs.

3. An intermediate result that occurs during a de-/encryption which needs to be a function of the processed data d_i and a small part of the key, which is denoted as k^*.

4. The power consumption of the device has to depend on the processed data. We refer to the measured power trace that corresponds to the processed data d_i as $\mathbf{t_i} = (t_{i,1}, \ldots, t_{i,T})$, where T denotes the length of the traces. We obtain a trace for each run so this results in the power trace matrix $\mathbf{T}$ with size $D \times T$.

Detailed instructions on how to perform a DPA can be found in [MOP07]. The basic idea for a DPA is, to make a hypothesis about one ore more bits of the secret key. Therefore, we chose an intermediate function, which depends on the processed data d_i and a small part of the key k^*. Since the small part of the key is unknown, all possible values $\mathbf{k}^* = (k_1, \ldots, k_j, \ldots, k_K)$ for k^* have to be processed, where the index j denotes the j^{th} possible key and the index i denotes the i^{th} en-/decryption. Now the hypothetical intermediate values $v_{i,j}$ can be calculated for all D de-/encryption runs and for all K possible key hypotheses. This results in a matrix $\mathbf{V}$ with size $D \times K$, where the column j contains the intermediate results that have been calculated with the key hypothesis k_j. Now, we can derive the hypothetical power consumptions $\mathbf{H}$ from the hypothetical intermediate values via a power model. The better the power model matches the actual power consumption characteristics of our hardware device the less power traces are needed. The most common power models are:

- the bit model (LSB-Least Significant Bit):
$$h_{i,j} = \mathrm{LSB}(v_{i,j}), \tag{3.12}$$
- the Hamming-weight (HW) model:
$$h_{i,j} = \mathrm{HW}(v_{i,j}), \tag{3.13}$$
- the Hamming-distance (HD) model:
$$h_{i,j} = \mathrm{HD}(v_{i,j}, d_i), \quad \text{and} \tag{3.14}$$
- the zero-value model. Here we assume, that the power consumption for the data value 0 is lower than for all other data values:
$$h_{i,j} = \begin{cases} 0 & \text{if } v_{i,j} = 0 \\ 1 & \text{if } v_{i,j} \neq 0. \end{cases} \tag{3.15}$$

The final step is to compare the measured power traces $\mathbf{T}$ with our hypothetical model $\mathbf{H}$. In [KJJ99], the authors measure correlations indirectly by computing the *difference of means*. In this thesis, we describe how the attack is performed using the *Pearson correlation coefficient*. A detailed description is in the Appendix. We estimate each value $r_{i,j}$ with

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \bar{h_i}) \cdot (t_{d,j} - \bar{t_j})}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h_i})^2 \cdot \sum_{d=1}^{D}(t_{d,j} - \bar{t_j})^2}} \quad i = 1, \ldots, K, \quad j = 1, \ldots, T. \tag{3.16}$$

With the resulting matrix $\mathbf{R}$ of correlation coefficients, we can determine if the hypothetical power consumption $\mathbf{H}$ and the measured power traces $\mathbf{T}$ are correlated. In Equation (3.16), $\bar{h_i}$ and $\bar{t_j}$ denote the mean values of the columns h_i in $\mathbf{H}$ and t_j in $\mathbf{T}$. Every row in matrix $\mathbf{R}$ is the correlation coefficient over time for a specific key hypothesis. If the key hypothesis ("key guess") was wrong, then the hypothetical power consumption and the power trace should be uncorrelated; i. e., we see just noise. If the key guess is correct, we should see a peak in the correlation curve.

A second order DPA attack considers multiple points simultaneously within the same power trace t. These points belong to two different intermediate values, e. g., a mask value m and the hidden intermediate value $d \oplus m$. The knowledge of either m or $d \oplus m$ alone is not of any use to the attacker. But if the attacker correlate both values, he can gain some information on d. A detailed description of higher order DPA can be found at [JPS05].

Chapter 4

Side-Channel Resistant AES Implementation

A main criteria for the feasibility of a power analysis attack is, that the part of the cipher which is attacked only depends on a small amount of key bits. AES offers exploitable functions like the first and the last *AddRoundKey*. They both depend only on a small amount of key bits. AES[1] provides another interesting characteristic which makes a power analysis attack even more applicable: The non-linear *SubBytes* transformation. This is because this transformation is non-linear and therefore if there is one bit wrong in the key hypothesis several bits of the output differ from the calculated intermediate result. On average, half of the bits calculated based on an incorrect key hypothesis differ from the actual intermediate result – independently from the number of incorrect bits in the key hypothesis. This improves the ability of finding the right key during a SCA enormously. Figure 4.1 depicts the initial *AddRoundKey* operation (XOR) before the first round and the first *SubBytes* transformation for the byte $s_{0,0}$ of the State matrix in round one. So an important task will be to protect

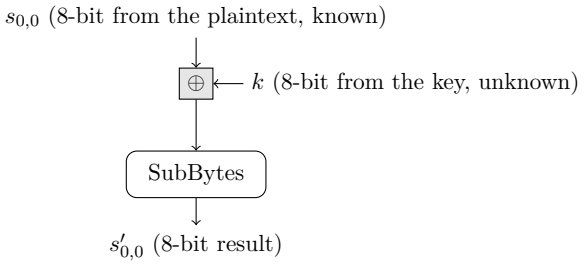

Figure 4.1: Initial AddRoundKey followed by the first SubBytes transformation in round one for one byte of the State matrix

the *SubBytes* transformation in the side-channel resistant implementation.

4.1 Random Bits

All following countermeasures need random bits. The *Bundesamt für Sicherheit in der Informationstechnik* (BSI) describe four classes for the quality of deterministic random number generators in [BSI01]. From the "weak" class K1, where the random numbers containing no identical consecutive elements with a high probability, to the "strong" class K4, where it should be impossible for an attacker to calculate or guess from an inner state of the generator any previous numbers in the sequence of random numbers.

1 like other ciphers with a diffusion step, for example DES

To mask the *SubBytes* transformation, it is very important to generate high class random numbers for the mask values, because if the distribution of the random mask has a bias with respect to the uniform distribution, the masked value leaks information, which can be exploited during a DPA, see [CGPR08].

So the random number generator should at least fulfill the requirements of a K2 random number generator for the masked values. This class ensures, that the random numbers passed statistical tests and are uniformly distributed. For generating those qualitative good random values, the IAIK suggests the Grain-80 stream cipher, which can also be used as random number generator. The Grain-80 is optimized for hardware implementations. In practice the AES implementation will get the random numbers from outside (a hardware random number generator), Grain-80 is only be used to generate random numbers for the final measurements on the development board.

4.2 Countermeasure: Masking

During a power analysis attack, the attacker has to make a hypothesis about the power consumption of the device performing the attacked operation. Our goal is, to make the real intermediate values of the executed transformation independent from their power consumption. This can be realized by masking them. With masking we conceal every intermediate value with a random value m, the so-called mask. The attacker does not know the masking values and thus cannot compute meaningful hypothesis on the power consumption. As a result, the side-channel leakage of all intermediate, key-dependent values, does not correlate with the corresponding unmasked value. We can easily reproduce the propagation of the mask throughout the linear parts of AES in order to remove the mask at the end of the cipher. For the non-linear parts, namely the *SubBytes* transformation, we require considerable more effort for the mask correction after this transformation.

In the following subsections, we will discuss the different masking schemes in terms of space consumption, security, and speed. For all masking schemes, the runtime of the key scheduling is excluded. The 176-byte RAM consumption for the expanded round key is included in the estimated RAM value.

4.2.1 Additive Masking

Additive masking masks a value x with a random value m by adding (XORing) them, $x_m = x \oplus m$. The mask can be removed by adding the mask value again, $x = x_m \oplus m$. This is an easy but very efficient masking scheme, since the masked value x_m is not distinguishable from a random value if the mask value m is a random value.

The "problem" with AES is, that the *SubBytes* transformation is a non-linear operation, which means $SubBytes(s_{i,j} \oplus m) \neq SubBytes(s_{i,j}) \oplus SubBytes(m)$. So additive masking cannot be easily applied "out of the box". The mask value m cannot be easily removed from the intermediate value $s'_{i,j}$ after the transformation without modifying the *SubBytes* transformation.

For additive masking, the whole lookup table (S-box) has to be recomputed. The input mask m masks the index and the output value mx masks the actual data according to Algorithm 3. This masking method was published first in [Mes00].

Algorithm 3: Computation of the masked AES SubBytes transformation as proposed in [Mes00]

Input: m, mx
Output: MaskedSubBytes($s_{i,j} \oplus mx$)=SubBytes($s_{i,j}$) $\oplus\, m$
for $i = 0$ **to** 255 **do**
    MaskedSubBytes($i \oplus mx$)=SubBytes(i) $\oplus\, m$;
end

With one mask value pair m and mx for all intermediate values, we need additional 256 bytes RAM for the new masked S-box and two random bytes for the input mask m and the output mask mx. With the expanded key that makes 176 bytes + 256 bytes + 2 bytes = 434 bytes.

The precalculation is made up of 256 additions (XOR) to mask the table index, 256 table lookups (TLU) to read the unmask S-box values from ROM, 256 additions (XOR) to mask the table entries and, finally, 256 write operations to store the masked table in RAM. This takes 256 cc + 2 · 256 cc + 256 cc + 256 cc = 1280 cc for computing the new S-box.

With four mask value pairs a whole column of the State matrix can be masked. The advantage is, that during the *MixColumns* transformation, which works on a whole column (4 bytes) of the State matrix, all intermediate values are mask with different masks. Therefore we need eight random bytes and 4 · 256 bytes = 1024 bytes for the new lookup tables, which results (with the expanded key) in 176 bytes + 8 bytes + 1024 bytes = 1208 bytes RAM. The precalculation overhead for four different masks leads to 4 · 1280 cc = 5120 cc for computing the new S-boxes. We can assume the runtime of the normal *SubBytes* transformation, since during an AES encryption, we just have to index the correct masked table for the TLU. In addition we have to mask the intermediate value from the State matrix with the input mask before and with the output mask after the transformation. That makes additional 2 cc per byte of the State matrix and 2 · 160 cc = 320 cc for the whole encryption. Summarizing, this scheme needs for all ten rounds 320 cc + 1280 cc = 1600 cc with one masked S-box and 320 cc + 5120 cc = 5440 cc with four masked S-boxes.

To mask the 32-bit T-tables implementation we would require 32-bit mask values, since we now operate on whole rows, instead of bytes from the State matrix. Masking the tables with one input– and one output–32-bit mask requires eight random bytes for the masks and 8192 bytes RAM for the eight masked T-tables (four for the first nine rounds and four for the last round). In total it makes (with the expanded key) 176 bytes + 8 bytes + 8192 bytes = 8376 bytes. The precalculation is made up of 256 additions (XOR) to mask the 32-bit table index, 256 table lookups (TLU) to read the unmask values from the tables (ROM), 256 additions (XOR) to mask the table entries and finally 256 write operations to store the masked table 32-bit values in RAM. This costs us 256 cc + 2 · 256 cc + 256 cc + 256 cc = 1280 cc for the precalculation of the masks and in total 520 cc + 1280 cc = 1800 cc for the whole AES.

Masking all four columns of the State matrix with the T-tables implementation would require 8 · 4 bytes = 32 random bytes for the masks and 4 · 8192 bytes = 32768 bytes for the masked tables. In total that makes with the expanded key 176 bytes + 32 bytes + 32768 bytes = 32972 bytes RAM. The precalculation of the tables take 4 · 1280 cc = 5120 cc and in total 520 cc + 5120 cc = 5640 cc for the whole AES.

4.2.2 Transformed Masking Method

In [AG01], Akkar and Giraud do not use a lookup table for the *SubBytes* transformation. Instead, they calculate the *SubBytes* transformation. Thereby, they mask the intermediate value $s_{i,j}$ multiplicative during the inversion part of the *SubBytes* transformation.

Figure 4.2 shows the masked version of the *SubBytes* transformation where a modified inversion function is used, which conserves the mask value. We will discuss the realization of this modified inversion in the following.

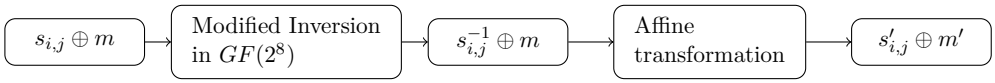

Figure 4.2: SubBytes transformation with masking countermeasure

The idea of *Transformed Multiplicative Masking* (TMM) is to add a new multiplicative mask $Y_{i,j}$ to the intermediate value $s_{i,j}$ and remove the additive mask $M_{i,j}$. The intermediate value is now masked with only the multiplicative mask. The advantage of the multiplicative mask is, that this mask can be removed after the inversion of the masked intermediate value easily by inverting the multiplicative mask $Y_{i,j}$ separately and then multiply that inverted mask with the inverted masked value. In Figure 4.3 we see on the left side, how the transition between additive and multiplicative masking is done. After the inversion, we can add the additive mask again and then remove the multiplicative mask. As result we get the inverse of the intermediate value $s_{i,j}$ additively masked with the mask $M_{i,j}$. This countermeasure is very costly since it contains an additional inversion and four multiplications in GF(2^8). A multiplication is marked with a dark gray color in Figure 4.3.

If we calculate the multiplication with the *binary multiplication method*, as it is suggested in [FIP01], we need for the multiplication at average 16 shift operations and 8 XOR operations, which makes $16\,\text{cc} + 8\,\text{cc} = 24\,\text{cc}$.

For an inversion of one byte from the State matrix with this method, we need four multiplications, two XOR, and two inversions (TLU). This costs us $4 \cdot 24\,\text{cc} + 2 \cdot 1\,\text{cc} + 2 \cdot 2\,\text{cc} = 102\,\text{cc}$. To invert the $10 \cdot 16\,\text{bytes} = 160\,\text{bytes}$ for all ten rounds, we need $160 \cdot 102\,\text{cc} = 16320\,\text{cc}$.

Akkar and Giraud give a cycle count of 293500 cc per encryption, which is about three times more than in the unprotected implementation we have seen in Section 2.3, which was 90500 cc.

In [TSG02], Trichina *et al.* have published a simplified variant of the algorithm which is shown in Figure 4.3 on the right hand side. The scheme is equivalent to the original masking scheme but the authors here use the same mask value for the multiplicative masking (Y=M).

For an inversion of one byte from the State matrix with this simplified method, we need only two multiplications, two XOR, and one inversion (TLU). This costs us $2{\cdot}24cc + 2{\cdot}1\,\text{cc} + 1{\cdot}2\,\text{cc} = 52\,\text{cc}$. To invert the 160 bytes for all ten rounds, we need $160 \cdot 52\,\text{cc} = 8320\,\text{cc}$.

The authors in [TSG02] give us no information about the cycles or memory consumption in their paper, so we will compare it to the Akkar/Giraud version. Since it saves the half on multiplications and inversion during one *SubBytes* transformation, which are the expensive and time consuming functions, we assume, that this implementation needs half of the cycles, the Transformed Masking Method from Akkar/Giraud would need.

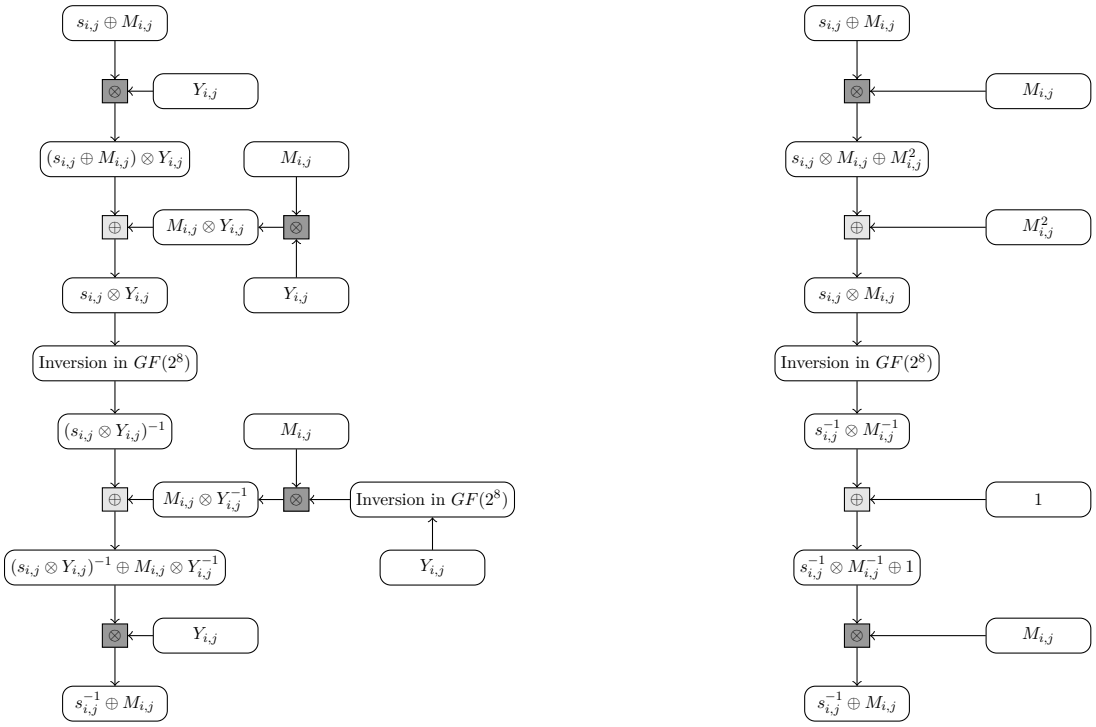

Figure 4.3: Multiplicative (left), Source: [AG01], and Simplified Multiplicative (right), Source: [TSG02](right), Masked inversion in SubBytes

Embedded Multiplicative Masking

Both, the normal and the simplified transformed masking method mask only *non-zero values*. If a key byte equals a data byte, then the result of *AddRoundKey* is zero. This fact can be exploited in a special (first-order) differential side-channel attack, the *zero-value attack*. In [OMP04] it is shown, that around 64 times more measurements are needed in a zero-value attack than in a standard differential side-channel attack. This indicates that they must be avoided because they still pose a serious practical threat.

The idea from [GT03] is to embed the finite field $GF(2^8) \simeq GF(2)[X]/P(x)$, $P(X) = (x^8+x^4+x^3+x+1)^2$ into the larger ring $R = GF(2)[x]/(P(x) \cdot Q(x))$. $Q(x)$ has degree k and needs to be irreducible over $GF(2)$ and coprime to $P(x)$. We define $\rho : GF(2)[x]/P(x) \rightarrow R$:

$$\rho(U) = U + R \times P \tag{4.1}$$

The mask $R(x)$ is a random polynomial $\neq 0$ of degree smaller than k. We define a mapping F on R, $F : R \rightarrow R$ with $F(U) = U^{254}$ which coincides with the inversion on $GF(2)[X]/P(x)$[3]. 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 random, the masking will not allow to reveal information even if $X = 0$. This zero value gets mapped to one of 2^k random values. This is a mathematically elegant way to bypass the zero value problem

2 P corresponds in our notation to the irreducible polynomial m. However, we stick to the notations of the paper [GT03] in this paragraph.

3 Fermat's Little Theorem

but it leads to very costly implementations. According to [BGK04], this leads in at least 731 AND and 766 XOR operations, including the transformations to the new field representation and back. For our cycle count this means that we need at least $731\,\text{cc} + 766\,\text{cc} = 1497\,\text{cc}$ for one byte of the State matrix and $160 \cdot 1497\,\text{cc} = 239520\,\text{cc}$ for all ten rounds.

4.2.3 Perfectly Masking AES Inverter Against First-Order Side-Channel Attacks

Another way to protect the *SubBytes* function is presented in [BGK04]. The authors obtain a perfectly masked algorithm for AES by computing the multiplicative inverse in $\text{GF}(2^8)$:

$$s_{i,j}^{-1} = \text{INV}(s_{i,j}) = \begin{cases} s_{i,j}^{-1}, & \text{if } s_{i,j} \in \text{GF}(2^8) \\ 0, & \text{if } s_{i,j} = 0 \end{cases} \tag{4.2}$$

and masking all intermediate values.

They calculate the inverse $s_{i,j}^{-1} = \text{INV}(s_{i,j})$, as $y = s_{i,j}^{254}$ by using the Square-and-Multiply algorithm. We discussed the algorithm and the resulting side-channel leakage in Section 3.1. The goal is to protect the value $s_{i,j}$, the intermediate values, and the resulting inverse during the calculation.

Let r, r' be independently and uniformly distributed random masks and $u = s_{i,j}$. Now, the authors start with an additively masked value $x = u + r$ and want to calculate $y = \text{INV}(u) + r'$.

Let us recall the Square-and-Multiply algorithm with the exponent $e = 254$ and the given values:

Algorithm 4: Left-to-Right binary Square-and-Multiply Algorithm

Input: $x = (u + r), n = 255, e = \{\text{e}_\text{p}, \ldots, \text{e}_0\}_2 = \{11111110\}_2$
Output: $y = x^e \bmod n = x^{-1} \bmod n$

1 $y = 1$
2 **for** $k = p$ **down to** 0 **do**
3 $\quad y = y^2 \bmod n$
4 $\quad$ **if** $e_k = 1$ **then**
5 $\quad\quad y = y \cdot x \bmod n$
6 $\quad$ **end**
7 **end**

As we can see, the algorithm consists of two operations, a squaring in Step 3 followed by a conditional multiplication with a reduction of n in Step 5. These are the operations, which were modified by the authors of [BGK04] to ensure the side-channel resistance.

In the following two algorithms, the input $s_{i,j}$ becomes the additively masked input $u + r_{l,k}$, where r is the lth mask byte in Step k of Algorithm 4. Our goal is, to invert u. The auxiliary values f, v, w and t are needed in the algorithm.

Algorithm 5 describes the perfectly masked squaring (PMS) method. During the first step, the input $u^e + r_{1,k-1}$ is squared. In Step two and three, they change the old mask $r_{1,k-1}$ to the new mask $r_{1,k}$. The desired output is a squared input with a fresh mask $u^{2e} + r_{1,k}$.

In Algorithm 6, we see the perfectly masked multiplication (PMM) method. This algorithm gets two input values: The output of the previous step and a freshly masked value x'. In Step 1, the authors calculate the product of the two masked values. In Steps 2-5 the auxiliary

Algorithm 5: Perfectly Masked Squaring (PMS), Source: [BGK04]

Input: $x = u^e + r_{1,k-1}, r_{1,k}$
Output: $t_k = u^{2e} + r_{1,k}$

1	$f_k = x^2$;	$\{f_1 = u^{2e} + r_{1,k-1}^2\}$
2	$w_{1,k} = r_{1,k-1}^2 + r_{1,k}$;	{auxiliary term to correct f_k}
3	$t_k = f_k + w_{1,k}$;	$\{t_k = u^{2e} + r_{1,k}\}$

terms are calculated. The last part of the algorithm, Step 6 and 7 removes the disturbing parts of the masked product by simply adding up the two auxiliary terms $w_{1,k}$, $w_{2,k}$ and f_k.

Algorithm 6: Perfectly Masked Multiplication (PMM), Source [BGK04]

Input: $x = u^e + r_{1,k-1}, x' = u + r_{2,k}, r_{1,k-1}, r_{1,k}, r_{2,k}$
Output: $t_k = u^{e+1} + r_{1,k}$

1	$f_k = x \cdot x'$;	$\{f_k = u^{e+1} + u^e \cdot r_{2,k} + u \cdot r_{1,k-1} + r_{1,k-1} \cdot r_{2,k}\}$
2	$v_{1,k} = x' \cdot r_{1,k-1}$;	$\{v_{1,k} u \cdot r_{1,k-1} + r_{1,k-1} \cdot r_{2,k}\}$
3	$v_{2,k} = v_{1,k} + r_{1,k}$;	$\{v_{2,k} = u \cdot r_{1,k-1} + r_{1,k-1} \cdot r_{2,k} + r_{1,k}\}$
4	$w_{1,k} = v_{2,k} + r_{1,k-1} \cdot r_{2,k}$;	$\{w_{1,k} = u \cdot r_{1,k-1} + r_{1,k}\}$
5	$w_{2,k} = x \cdot r_{2,k}$;	$\{w_{2,k} u^e \cdot r_{2,k} + r_{1,k-1} \cdot r_{2,k}\}$
6	$t_{1,k} = f_i + w_{1,k}$;	$\{t_{1,k} = u^{e+1} + u^e \cdot r_{2,k} + r_{1,k-1} \cdot r_{2,k} + r_{1,k}\}$
7	$t_k = t_{1,k} + w_{2,k}$;	$\{t_k = u^{e+1} + r_{1,k}\}$

To calculate the masked inverse of one element with the secured Square-and-Multiply algorithm, we need seven PMS with additional PMM, both including a modular reduction step, and at last one PMS with a modular reduction. Every intermediate value of the PMS and PMM is hidden by a fresh mask. So we need $(7 \cdot 3) + 1 = 22$ random masks for the inversion of one element of the State matrix. For a whole AES en-/decryption we need $160 \cdot 22 = 3520$ random mask values each with the size of one byte. In the case of a first order side channel resistance, the authors point out in [BGK04, Section 5.4], that the number of random masks can be reduced to only three masks (r_1, r_2, r_3), by using the same masks in each step k.

To calculate the clock cycles for the inversion, we recall the clock cycle for: addition (1 cc), multiplication (24 cc) and squaring (2 cc). By counting the operations during the algorithm we get: $2 \cdot 2\,\text{cc} + 2\,\text{cc} = 6\,\text{cc}$ (two squaring and two additions), for one PMS, and $4 \cdot 24\,\text{cc} + 2 \cdot 1\,\text{cc} = 98\,\text{cc}$ (four multiplications and four additions) for one PMM operation.

So we need:

- $(7 \cdot 6)\,\text{cc} = 42\,\text{cc}$ for the PMS operation
- $(7 \cdot 98)\,\text{cc} = 686\,\text{cc}$ for the PMM operation
- 3 cc for loading the random intermediate mask $r_{1,k}, r_{1,k-1}, r_{2,k}$,
- 1 cc for loading the masked intermediate value from RAM, and
- 1 cc for saving the result.

In total we need $(42 + 686 + 3 + 1 + 1)\,\text{cc} = 733\,\text{cc}$ for the inversion of one element from the State matrix. For a whole AES en-/decryption we need $160 \cdot 733\,\text{cc} = 117280\,\text{cc}$ for the inversion part of the *SubBytes* transformation with three masks.

4.2.4 Combined Masking in Tower Fields

In [OMP04] and [OMPR05], Oswald *et al.* present a masking scheme, which is based on the composite field arithmetic we discussed in Section 2.3.8. They call it *combined masking in tower fields* (CMTF). The idea is, that the inversion in $\text{GF}(2^2)$ is a linear transformation, which gives us the potential to track the changes which are applied to the mask value m during an inversion.

To securely calculate the inversion of the current intermediate byte $s_{i,j}$ from the State matrix, we first mask it additively with the mask m to obtain $a = s_{i,j} \oplus m$. Next we transform a to its composite field representation. Please recall, that every element of $\text{GF}(2^8)$ can be represented as a linear polynomial over $\text{GF}(2^4)$. The finite fields, which we will use are the same as before (Section 2.3.8):

$$GF(2^8) \simeq GF(2)[x]/(x^8 + x^4 + x^3 + x + 1), \tag{4.3}$$

$$GF(2^4) \simeq GF(2)[x]/(x^4 + x + 1). \tag{4.4}$$

Note that operations $\oplus$ and $\otimes$ which we use here, are for coefficients from $\text{GF}(2^4)$. We have introduced them in Section 2.3.8.

After the mapping, the value that needs to be inverted is represented by $(a_h \oplus m_h)x + (a_l \oplus m_l)$ instead of $a_h x + a_l$ with $a_h, a_l, m_h, m_l \in GF(2^4)$. Both values, a_h and a_l, are now additively masked. It is easier to follow the masked inversion if we recall the normal inversion in the composite field:

$$(a_h x \oplus a_l)^{-1} = a'_h x \oplus a'_l \tag{4.5a}$$

$$a'_h = a_h \otimes d^{-1} \tag{4.5b}$$

$$a'_l = (a_h \oplus a_l) \otimes d^{-1} \tag{4.5c}$$

$$d = (a_h^2 \otimes \{\text{E}\}_{16}) \oplus (a_h \otimes a_l) \oplus a_l^2 \tag{4.5d}$$

We can calculate the masked inverse, by following the steps from (4.5a) to (4.5d), but this time we use the masked values:

$$((a_h \oplus m_h)x + (a_l \oplus m_l))^{-1} = (a'_h \oplus m'_h)x + (a'_l \oplus m'_l) \tag{4.6a}$$

$$a'_h \oplus m'_h = a_h \otimes d^{-1} \oplus m'_h \tag{4.6b}$$

$$a'_l \oplus m'_l = (a_h \oplus a_l) \otimes d^{-1} \oplus m'_l \tag{4.6c}$$

$$d \oplus m_h = a_h^2 \otimes \{E\}_{16} \oplus a_h \otimes a_l \oplus a_l^2 \oplus m_h \tag{4.6d}$$

Now we need to securely calculate the inverse for $d \oplus m_h, d \in GF(2^4)$. So far, we have only shifted the problem from the bigger field down to the smaller one. Because $\text{GF}(2^8)$ is a tower field over $\text{GF}(2^2)$, we can shift the computation of the masked inversion for d down to $\text{GF}(2^2)$ the same way as we did it in (4.6a) to (4.6d) with the corresponding field operations. In this field, the inversion is a linear operation and we keep track of the mask value. The inversion operation preserves the masking value, because $(d \oplus m)^{-1} = (d \oplus m)^2 = d^2 \oplus m^2$ in $GF(2^2)$.

This approach has been presented in [OMPR05] to work in tower fields which are applicable on hardware implementations but inefficient to implement in software on a 32-bit processor, since during the field transformations we work on every individual bit of all bytes in the State matrix. Instead, all transformations above can be precomputed and stored in tables. This method will be discussed in the next subsection.

Combined Masking in Tower Fields with Precomputed Tables

To compute the masked inversion in GF(2^8), Schramm *et al.* presented in [OS05] a method which is based on the inversion in composite fields, as we discussed it in the last section. The difference is, that the calculations of the intermediate values are mapped to a sequence of table lookups. Therefore the authors first map the masked intermediate value $a = s_{i,j} \oplus m$ down to its field representation with coefficients from GF(2^4) as done before.

With the use of the following four tables:

$$\begin{aligned} T_{d_1} &: ((x \oplus m), m) \mapsto x^2 \times \{\mathtt{E}\}_{16} \oplus m \\ T_{d_2} &: ((x \oplus m), (y \oplus m')) \mapsto ((x \oplus m) \oplus (y \oplus m')) \times (y \oplus m') \\ T_{m} &: ((x \oplus m), (y \oplus m')) \mapsto (x \oplus m) \times (y \oplus m') \\ T_{inv} &: ((x \oplus m), m) \mapsto x^{-1} \oplus m \end{aligned}$$

which can be precalculated and stored in ROM, the masked intermediate values of the formula (4.6b) to (4.5d) are calculated. Note, that all tables take two elements of $GF(2^4)$ as inputs and give an element of $GF(2^4)$ as output.

$$\begin{aligned} d \oplus m_h = T_{d_1}(a_h \oplus m_h, m_h) \oplus T_{d_2}((a_h \oplus m_h), (a_l \oplus m_l)) \\ \oplus T_m((a_h \oplus m_h), m_l) \oplus T_m((a_l \oplus m_l), m_h) \oplus T_m((m_h \oplus m_l), m_l). \end{aligned} \quad (4.7)$$

We compute the masked inversion of d with the following table lookup:

$$d^{-1} \oplus m_h = T_{inv}(d \oplus m_h, m_h). \quad (4.8)$$

Now we can compute $a'_h \oplus m'_h$ and $a'_l \oplus m'_l$ from (4.6b) and (4.6c) the following way:

$$\begin{aligned} a'_h \oplus m'_h = T_m(a_h \oplus m_h, d^{-1} \oplus m_l) \\ \oplus m_h \oplus T_m(d^{-1} \oplus m_l, m_h) \oplus T_m(a_h \oplus m_h, m_l) \oplus T_m(m_h, m_l), \end{aligned} \quad (4.9)$$

and

$$\begin{aligned} a'_l \oplus m'_l = T_m((a_l \oplus m_l), (d^{-1} \oplus m_h)) \\ \oplus m_l \oplus T_m(d^{-1} \oplus m_h, m_l) \oplus T_m(a_l \oplus m_l, m_h) \oplus f_{a_h} \oplus m_h \oplus T_m(m_h, m_l). \end{aligned} \quad (4.10)$$

To map $GF(2^8)$ elements to $GF(2^4) \times GF(2^4)$ elements and vice versa, two additional tables with a size of 512 bytes are needed.

As the authors state and we can count above, the total costs of a masked inversion this way are 14 table lookup operations and 15 XOR operations for an inversion, which results in $14 \cdot 2\,\text{cc} + 15\,\text{cc} = 43\,\text{cc}$. For an AES encryption this results in $160 \cdot 43\,\text{cc} = 6880\,\text{cc}$ additional for the new masked *SubBytes* transformations.

To store the six tables we need $4 \cdot 256\,\text{bytes} + 2 \cdot 256\,\text{bytes} = 1536\,\text{bytes}$ of ROM.

4.2.5 Masking the *MixColumns* Transformation

The *MixColumns* transformation can leak timing informations because it uses the *xtimes* operation. *Xtimes* contains a conditional reduction by $\{\texttt{11B}\}_{16}$ if the result of the operation is larger than eight bits, see Section 2.2.

To prevent the timing information leakage and power leakage, the State $\mathbf{S}$ has to be additively masked with the mask $\mathbf{M}$ before the transformation, $\mathbf{S_M} = \mathbf{S} \oplus \mathbf{M}$, where $\mathbf{M}$ is a 4×4 matrix created of four random bytes $M_1, \ldots, M_4$, see (4.11). After the transformation ($\mathbf{S'_M} = \mathit{MixColumns}(\mathbf{S_M})$) the mask $\mathbf{M}$ can be removed from the transformed State $\mathbf{S'_M}$ by adding the transformed mask $\mathbf{M'} = \mathit{MixColumns}(\mathbf{M})$, so $\mathbf{S'} = \mathbf{S'_M} \oplus \mathbf{M'}$.

$$\mathbf{M} = \begin{pmatrix} M_1 & M_2 & M_3 & M_4 \\ M_1 & M_2 & M_3 & M_4 \\ M_1 & M_2 & M_3 & M_4 \\ M_1 & M_2 & M_3 & M_4 \end{pmatrix} \tag{4.11}$$

Masking the *MixColumns* transformation needs one additional *MixColumns* transformation (84 cc) to get **M'** if the same mask $\mathbf{M}$ is used to mask all *MixColumns* transformations during an whole AES run. In addition, four random bytes are needed for the mask $\mathbf{M}$, and 16 bytes are needed for the transformed mask $\mathbf{M'} = \mathit{MixColumns}(\mathbf{M})$.

4.3 Countermeasure: Randomization

Randomizing the execution of the algorithm provides additional resistance against power analysis attacks. Herbst *et al.* suggest in [HOM06] two ways to introduce randomness in the execution of the algorithm. They propose to shuffle the sequence of operations which work on the elements of the State matrix and to add dummy rounds (or parts of a round) during the normal ones.

4.3.1 Shuffling

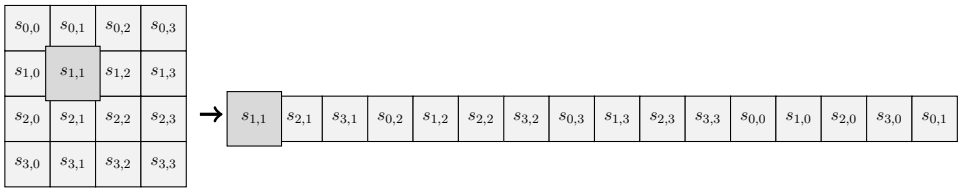

Figure 4.4: Shuffled AES-State

Figure 4.4 shows, we first randomly choose a starting column and in that column a random value. Then we proceed with the next element. As we work on each intermediate value of the State matrix, this version of shuffling is applicable on 8-bit implementations. With shuffling, the probability that one specific intermediate value is processed at a specific point in time is 1/16. Theoretically, we can shuffle the whole State matrix with 16! possibilities. But that would not increase the security against a first-order DPA with the same factor since here it

is important that the operations with the intermediate values do not occur on the same fixed time.

The randomization can be broken with windowing, published by Clavier *et al.* [CCD00]. Windowing can be seen as performing multiple second-order DPA attacks in parallel and then combining the power traces for the mask processing with each of the points in time, where the targeted masked values can occur. So shuffling all 16! possibilities would not bring us more security but would increase the implementation costs. For a successful side-channel attack 16 times more traces are needed with this countermeasure than with an unprotected implementation.

4.3.2 Execution of Dummy Operations

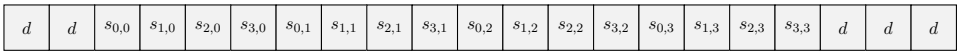

Figure 4.5: AES-State with dummy values

Another way to increase the randomization is to add dummy intermediate values d to the State matrix which will be operated on. Those operations must not be distinguishable from real operations of the algorithm. The problem with added wait states or, for example, just added random NOPs (No Operation) is that they can be easily removed by analyzing a single power trace. The authors in [HOM06] therefore suggest to insert dummy rounds or parts of a round on some dummy State bytes, see Figure 4.5. The combination of shuffling and dummy operations make it very difficult for the attacker to locate the attack point in the first two and the last two rounds. These are the rounds, which need our attention, because they are vulnerable during a known/chosen plaintext/ciphertext attack. After these rounds, every value from the State has been subjected to three *AddRoundKey* transformations and depend on sufficient many key bytes, to prevent an DPA attack [HOM06]. The probability, that a certain intermediate value occurs on a specific point in time is $p = 1/(D + 1)$, where D is the number of additional dummy cycles. If we chose to insert 15 dummy operations, the probability is 1/16, which would increase the number of needed traces for a successful SCA also by 16 times. Note, that we have to see the normal consecutive State bytes as *one* value compared to the dummy bytes, to calculate this probability.

With D additional dummy rounds and shuffling, which is recommended by the authors, the number of required traces for a successful mounted SCA increases by $(D + 1) \cdot 16$. With, e. g., $D = 5$ dummy rounds randomly inserted at the start and the end of the algorithm, 96 times more measurements would be required for a successful DPA attack.

Figure 4.6 denotes the execution on dummy bytes d and shuffled execution of the State matrix.

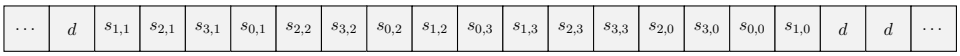

Figure 4.6: AES-State with dummy values and shuffled AES-State

The number of added dummy operations D is a fixed value. The random number r, $0 \leq$

$r \leq D$ partitions the D dummy operations into two parts. The dummy operations from 0 to $r-1$ will be executed before and the dummy operations from r to $D-1$ will be executed after the normal operations, like it is depicted in Figure 4.5 for $r = 2$ and $D = 5$.

Table 4.1 denotes how many random bits are required for the shuffling and the dummy cycles countermeasure.

Algorithm	**No. random bits**	**Usage**
shuffling	4	defines the entry point in State matrix
dummy operations	$\lceil \log_2(D) \rceil$	number of dummy operations before/after actual operation

Table 4.1: Number of random values required for secured implementation

4.4 Comparison

In this chapter we gave a short overview about the discussed (low-cost) software countermeasures against timing and power attacks. Now we estimate the number of clock cycles for a protected AES implementation with the selected countermeasure based on our 32-bit optimized implementation with a clock cycle count of 1732 cc. The clocks cycles for the affine transformation (Section 2.3.3) with 6560 cc are added to those schemes, which have to calculate it. We get the following runtimes estimated for the countermeasures:

- Additive Masking: 1732 cc + 1600 cc = 3332 cc,
- Transformed Masking: 1732 cc + 6560 cc + 16320 cc = 24612 cc,
- Transformed Masking simpl.: 1732 cc + 6560 cc + 8320 cc = 16612 cc,
- Embedded Masking: 1732 cc + 6560 cc + 239520 cc = 247812 cc,
- Perfect Masking: 1732 cc + 6560 cc + 117280 cc = 125572 cc,
- CMTF with TL: 1732 cc + 6560 cc + 6880 cc = 15172 cc.

Table 4.2 contains the estimated memory consumption (ROM and RAM), the number of random bits, and the runtime in clock cycles (cc) for the discussed masking methods. Note that we are only interested in a rough estimate to select the most appropriate countermeasure. The value on the left side denotes the estimated result for using one random byte to mask all 16 elements of the State matrix. The value on the right side is the estimated value using four random bytes to mask a whole column of the State matrix. All RAM estimates include the 176-byte expanded key array for the key scheduling.

With 3332 cc and 434 bytes RAM, Additive Masking is applicable when used with one mask. With four different masks the RAM consumption of 1208 bytes, which is mainly caused by the four masked S-boxes, is too big for our needs. The runtime 5440 cc + 1732 cc = 7172 cc is still acceptable.

In a 32-bit T-tables implementation, which is Additively Masked with one mask pair, the RAM consumption of 8376 bytes and 32976 bytes for four masks is also too big.

The Transformed Masking and simplified Transformed Masking Method both multiplicatively mask an element of the State matrix during the *SubBytes* transformation. The simplified Transformed Masking Method is the improved one of the Transformed Masking Method, in terms of needed operation and random bytes for masking. It is very interesting regarded to the RAM consumption of 180 bytes. Compared to the Additive Masking implementation, the runtime of 16612 cc is also acceptable. A big pro of the Transformed Masking approach is the constant runtime, which is independent from the number of mask values.

The Perfect Masking in line six needs 179 bytes RAM (three for the masks and 176 for the expanded key). Since it calculates the inversion transformation and the affine transformation on the fly, it has a very big runtime, which is the main criteria against it.

No.	**Masking Method**	**ROM** (bytes)	**RAM** (bytes)	**Encryption** (clock cycles)	**No. of random bits**
1	unmasked	256	176 / 176	1732 / 1732	0 / 0
2	Additive Masking	256	434 / 1208	3332 / 7172	16 / 64
3	Transformed Masking	0	178 / 204	24612 / 24612	16 / 64
4	Transformed Masking simpl.	0	177 / 180	16612 / 16612	8 / 32
5	Embedded Masking	0	177 / 180	247812 / 247812	8 / 32
6	Perfect Masking	0	179 / 179	125572 / 125572	3 / 12
7	CMTF with TL	1536	177 / 180	15172 / 15172	8 / 32
8	T-tables, unmasked	8192	176 / 176	520 / 520	64 / 64
9	Additive Masking (T-tables)	8192	8376 / 32976	1800 / 5640	256 / 256

Table 4.2: Estimated memory consumption and cycle count for different AES implementations with masking countermeasure, one mask / four masks

The same constant runtime characteristic holds for the Combined Masking in Tower Fields in row seven. The big ROM consumption of 1536 bytes mainly results from a couple of constant tables, which are used for the transformation between the different representation systems and for calculating the masked inversion, see Section 4.2.4.

The big part of the runtime from the Additive Masking Method arises from the creation of the masked S-boxes. If these masked S-boxes have been created, the implementation is even as fast as the unmasked implementation since then, the transformation is only a table lookup. Transformed Masking and the improved versions of it would only pay off if a high number of random masks is used. But using a high number of masks does not lead to a practically safer implementation. This is because a higher order DPA works whenever the same mask values occur on two different points in time. Since all masks have to be created at a specific point in time and are used to mask an intermediate value at some point later, there are always two points in time that allow a higher order DPA attack.

We decided to use the Additive Masking Method and combine it with randomization (shuffling and dummy states). By using the masking approach, we fulfill the requirements needed against a first-order DPA. In addition, we will mask the *MixColumns* transformation to prevent timing analysis attacks and use masking and shuffling to increase the resistance to a second-order DPA.

4.5 Low-Cost SCA-Resistant AES Software Implementation

On the Applied Cryptography and Network Security Conference (ACNS'07), Tillich, Herbst and Mangard [THM07] presented a low-cost SCA-resistant AES implementation, which takes advantage of the masking, shuffling and execution of dummy cycle approach. In Figure 4.7, the program flow of the randomized and masked AES is depicted. The randomization areas are called Randomization Zone 1 and Randomization Zone 2. In these two zones the shuffling and the dummy operation approach is applied. Because the randomization is very costly, the authors keep the Randomization Zones as short as possible. During the Randomization Zones, the following operations are randomized: The *AddRoundKey* operation, since the 16 bytes of the State matrix **S** are processed independently. The same holds for the (masked) *SubBytes* transformation. During the *MixColumns* transformation, the sequence of the processed columns is randomized.

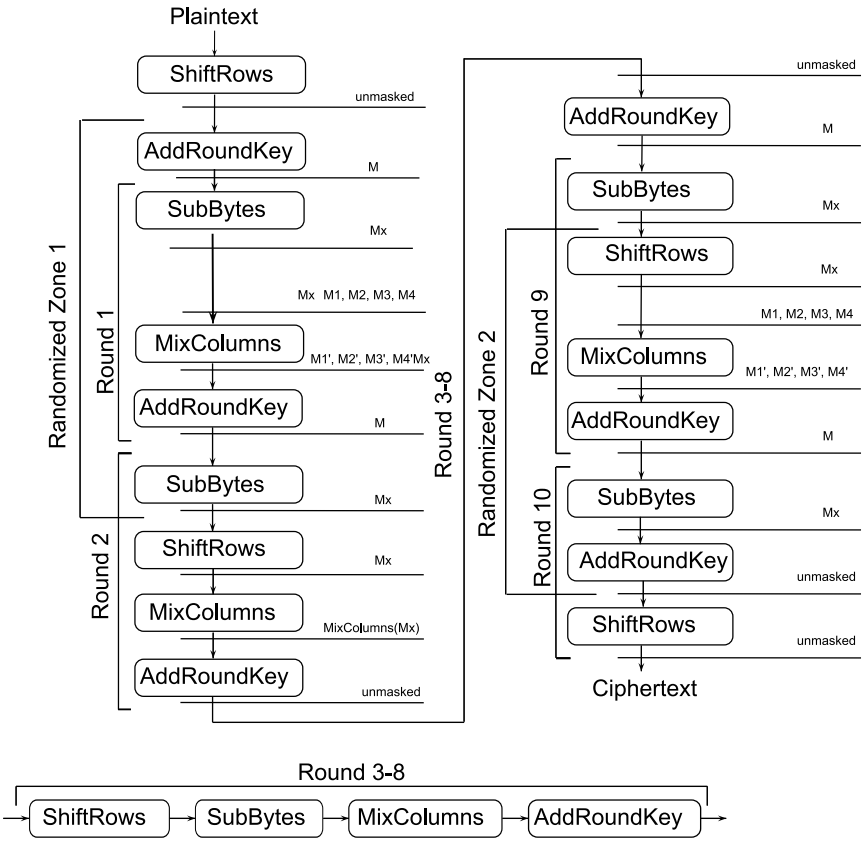

Figure 4.7: Program flow of a randomized AES implementation where all transformations are masked in the first and last two rounds. The current masks (M, M', M_1,...,M_4 and M'_1,...,M'_4) are depicted right of the State. Adapted from: [THM07].

The first two and the last two rounds of the AES implementation are additionally protected by masking. The current mask of the State **S** is shown on the right in Figure 4.7. The *SubBytes* transformation is masked additively with the mask byte M, see Section 4.2.1.

To protect the *MixColumns* transformation, the authors use four mask bytes, M_1 to M_4, see Section 4.2.5. Every mask byte M_1 to M_4 is used to mask the corresponding column

of the State **S**. To remove the four masks after the *MixColumns* transformation, the four mask bytes are also transformed, $(M_1', M_2', M_3', M_4') = MixColumns(M_1, M_2, M_3, M_4)$. To remask the State, the *AddRoundKey* transformation is used whenever possible. The *ShiftRows* transformation is excluded from the Randomization Zones because it works on the whole State matrix.

We will adapt this scheme. The Additive Masking implementation with one masked S-box needs needs 3332 cc. To calculate the four masks M_1', M_2', M_3' and M_4' we need one additional *MixColumns* transformation (84 cc). We implement one dummy operation to operate on one column of the State matrix. This is the smallest part on which all transformations can work and it fits to our 32-bit architecture. During one dummy operation the following transformations are executed `AddRoundKeyColumn, SubBytesColumn, MixColumn, AddRoundKeyColumn and SubBytesColumn`. This takes 4 cc + 20 cc + 21 cc + 4 cc + 20 cc = 69 cc. For shuffling, we only change the indices of the current column and current row. We think this runtime is negligible. So the protected implementation has a runtime of 3332 cc + 84 cc = 3416 cc with no dummy operations and $3416 + D \cdot 69$ cc with D dummy operations.

The Additive Masking scheme needs 256 bytes RAM for the masked S-box and 16 bytes for the four words $M1'$ to $M4'$. In addition six random bytes for the masked values M, M', M_1 to M_4 one random byte r to divide the number of dummy operations during the randomization zones, 16 random bytes for one dummy State and five random bytes for shuffling the five *AddRoundKey*, four *SubBytes* and three *MixColumns* transformations in the two randomization zones. That makes in total $(176 + 256 + 16 + 6 + 1 + 16 + 5)$ bytes = 476 bytes RAM.

Implementation	**Encryption** (clock cycles)
Unmasked	4427
Masked	8420
Masked and Randomized	11845+D · 240

Table 4.3: Comparison of an unmasked AES implementations with a masked and a masked and randomized AES implementation with D dummy operations on an 8-bit smart card processors (AVR), Source: [HOM06].

In Table 4.3 the execution time between an unmasked and their randomized AES implementation is compared by the authors of [HOM06]. The masked AES implementation is roughly two times slower than their referenced unmasked AES implementation. The randomized implementation takes about 11845 clock cycles when no dummy operations are added. This is about 2.7 times slower than the unprotected AVR-based AES implementation. With D dummy operations added, their randomized AES implementation needs $11845 + D \cdot 240$ clock cycles. Our estimated protected AES encryption increases by a factor of two.

Chapter 5

Target Platform: TriCore TC1796

To write an optimized AES implementation, it is important to understand the underlying microcontroller architecture. We will discuss the most important parts of the microcontroller in the following subsections and only mention those features of the TriCore, which are interesting for our work.

The TriCore TC1796 or variants thereof is a typical automotive embedded processor which is used, for example, in the current engine control units (ECU) MEDC17 by Robert Bosch GmbH. The TriCore TC1796 combines a reduced instruction set computing (RISC) processor, a digital signal processor (DSP), and on-chip memories as well as peripherals. It has a program flash memory of 2 Mbyte, a data flash of 128 Kbyte, and 136 Kbyte data memory and a 16 Kbyte instruction cache (ICACHE). The TriCore TC1796 has no data cache [Inf07, page 2-36].

The program flash and the data flash are connected through the SBD (System Bus Domain) to the CPU. Because the data is accessed through a bridge, it can be one order of magnitude slower than the local domain which is located nearest to the core [Inf02, page 30]. This means for our processor model, that the runtime can also differ by an order of magnitude because our program code and program data is located at the flash.

The chip provides a lot more interesting features, which are explained in the data sheet of the chip [Inf08a]. But first let us start with an overview of the development environment used.

5.1 Development Environment

For programming and compilation of our code, we use the Tasking VX tool set version 3.0r1. It offers a `C` compiler, linker, Assembler, and a simulator for the TriCore. The `C` compiler supports the ISO C99 standard.

At the Robert Bosch GmbH, the Hightech `GNU C` development environment is widely used and also used for developing TriCore applications. We write our code in such a way, that it compiles on both development environments.

We use the Universal Debug Engine (UDE) UAD2 to communicate with the evaluation board. The debugger is connected via USB to our PC. To flash our application to the development board and to debug it in real time, we use `UDE-Desktop` application. The development board is a `TriCore Evaboard TriBoard TC1796`.

5.1.1 Compiler

We write our implementation in C following the C99 standard to keep it as portable as possible. If some optimizations are only possible in assembler, we will write a TriCore optimized version and a generic version, separated by defines in the source.

Assembly instructions can be inserted with the keyword `__asm` in C source code. The C variables are passed as operand to the assembly code. The general syntax of the `__asm` keyword is:

```
__asm("instruction_template"
       [ : output_param_list
       [ : input_param_list
       [ : register_save_list]]] );
```

The `instruction_template` is the assembly instruction, that may contain parameter from the `input_param_list` or `output_param_list`. The `register_save_list` contains the name of registers which should not be used.

The following example multiplies two C variables and assigns the result to a third C variable. The `=d` denotes that a data register is necessary for output, and the `d` denotes that the values come from a data register.

```
1  int in1, in2, out;
2
3  void initreg( void )
4  {
5    __asm( "mul %0, %1, %2"
6                :"=d" (out)
7                :"d" (in1), "d" (in2));
8  }
```

The following sections list the built in functions with their respective prototypes and associated machine instructions. In case an assembler mnemonic is given, the builtin's semantic is the same (from the C programmer's point of view) as the semantic of the machine instruction. We do not repeat the instruction set manual here.

Please note, that the syntax of the Hightech GNU C compiler differs for the inline assembler. However, the compiler has built in functions for the assembler instructions which can be used instead.

To ensure the formal correctness of our code, we use `splint` (Secure Programming Lint). *Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes.*[1]

Because our implementation will be used in the automotive environment it should also be Motor Industry Software Reliability Association[2] (MISRA) conform. MISRA is a set of programming rules for secure programming in C.

For code formating we use `indent -no-tabs`. We will deliver our AES implementation as source code, which can be compiled as a library.

1 http://splint.org/
2 http://www.misra.org.uk/

5.1.2 Simulator

During the development process, we use the TriCore Instruction Set Simulator (TSIM) to simulate and debug our implementation. It is described in [Inf05] and perfectly integrated in our development IDE. The simulator has the disadvantage, that it does not simulate the pipelining right. Therefore, we will make the time measurements on the evaluation board.

5.1.3 Evaluation Board

We have the `TriBoard - TC1796 V4.1` which hosts the TriCore TC1796. This board is connected via a JTAG (Joint Test Action Group) interface to the UAD. The Debugger is connected via USB to our development PC.

We can debug the TriCore with the Universal Debug Engine (UDE). The UDE has a virtual console, where we can receive messages from our application on the TriBoard. To receive the messages during a run on the evaluation board, we have to set the define `CONSOLE_TYPE` in `BCLtypes.h` to `CONSOLE_TYPE_TASKING`. This causes the compiler to overwrite the standard `POSIX` read and write functions, which are called by the `ANSI-C` Library. The `stdin`, `stdout` and `stderr` are now piped through a JTAG interface and we see the output in the UDE-Desktop application.

5.1.4 The Multi-Core Debug System

In addition, we got the `TriBoard - TC1796.303`, which hosts the TC1796ED. With this board, we can do exact timing measurements without external influences. Figure 5.1 depicts the TriBoard and how the UAD is connected via the JTAG interface.

The TC1796ED provides a Multi-Core Debug System (MCDS) that can be used to trace the program, which is executed on the TC1796, in real-time. Its MCDS is integrated as a plug-in into the UDE-Desktop application. Figure 5.2 depicts the MCDS configuration dialog. The dialog contains 5 tasks which are needed to measure the runtime of the function `aes_Encrypt`. The first task configures the emulation device, there the timer resolution is set to 250 ns.

The next two tasks create one signal each. The first signal (sig_start) fires if the TriCore program counter (PC) reaches the start address of the function `aes_Encrypt`. The second signal (sig_stop) fires when the end of `aes_Encrypt` is reached. The following two tasks define the actions on the signals. The first action starts a timer, the second one stops the timer and stores its value for every execution of the function `aes_Encrypt`.

5.2 TriCore CPU

The TC1796 processor contains a TriCore 1 V1.3 central processing unit (CPU), with a maximum CPU frequency of f_{CPU}=150 MHz. The most interesting features of the CPU for us are the:

- 32-bit load/store architecture,
- 16-bit and 32-bit instructions for reduced code size,
- data formats: bit, byte (8-bit), half-word (16-bit), word (32-bit), double-word (64-bit),

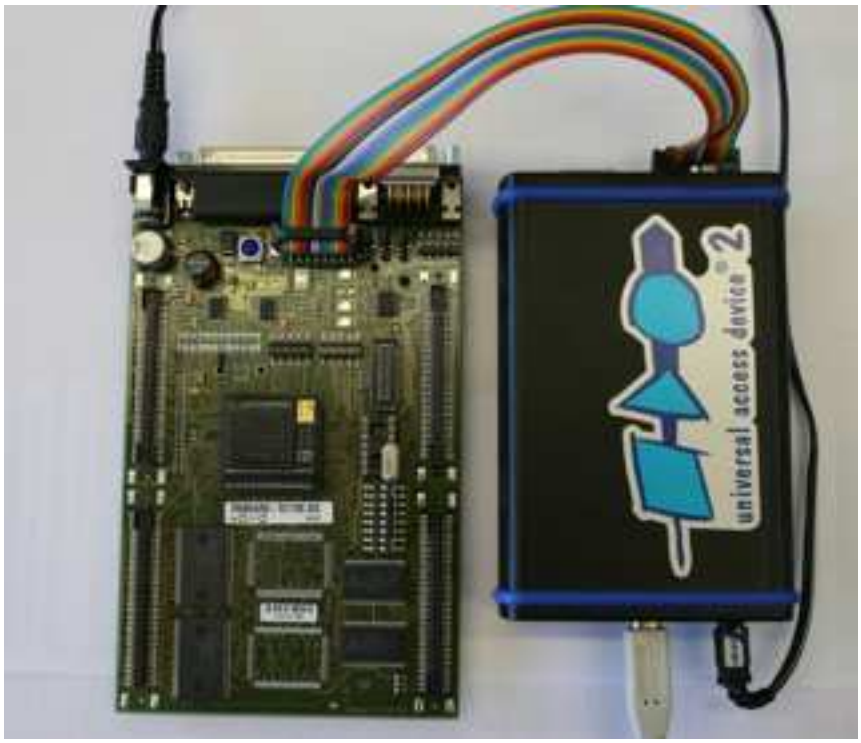

Figure 5.1: TriBoard – TC1796.303 on the left and AD2 on the right

- byte and bit addressing,
- little endian byte ordering,
- packed data.

This enumeration is taken from the user manual [Inf07, Section 2]. The CPU contains an instruction fetch unit, an execution unit, and a general purpose register file (GPR), namely the address and data registers.

The instruction fetch unit pre-fetches and aligns the incoming instructions. The execution unit of the TriCore contains three pipelines which work in parallel. This allows us to execute up to three instructions in one clock cycle. The execution unit contains the integer pipeline (IP), the load/store pipeline (LS), and the loop pipeline.

The integer pipeline and the load/store pipeline have the following four stages: fetch, decode, execute, and write-back. The loop pipeline has the two stages: decode and write-back. The execution unit can execute most operations we use in one clock cycle. Figure 5.3 illustrates the three pipelines of the execution unit.

The instruction fetch unit feeds the three pipelines. The fetch unit is able of issuing, in one cycle, an integer pipeline instruction to the integer pipeline and an immediately following LS instruction to the LS pipeline.

Our two main goals during our TriCore optimization strategy are to avoid pipeline stalls as much as possible and to take advantage of the parallel issue capabilities as mentioned above.

The CPU has 16 data registers (register D0 to D15) and 16 address registers (register A0 to A15). Each address and data register has a size of 32-bit. Respectively two 32-bit data

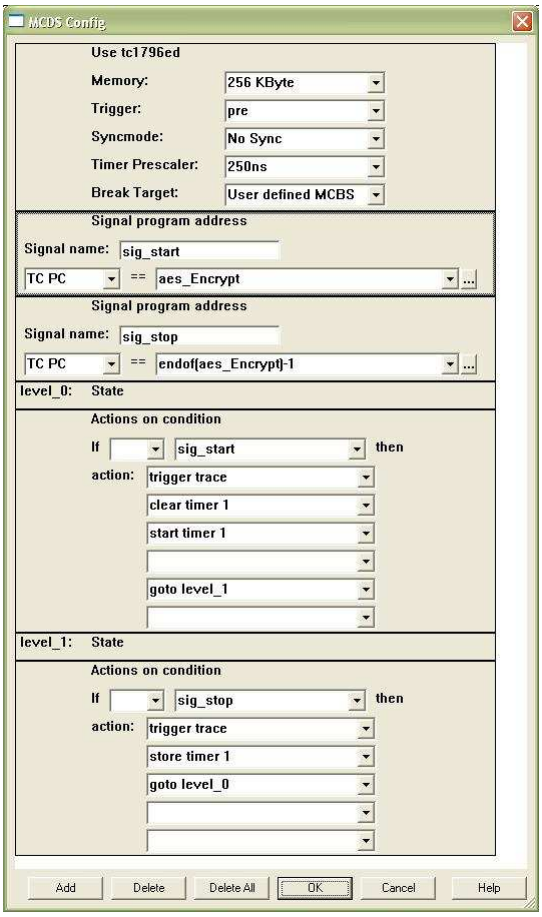

Figure 5.2: MCDS config dialog

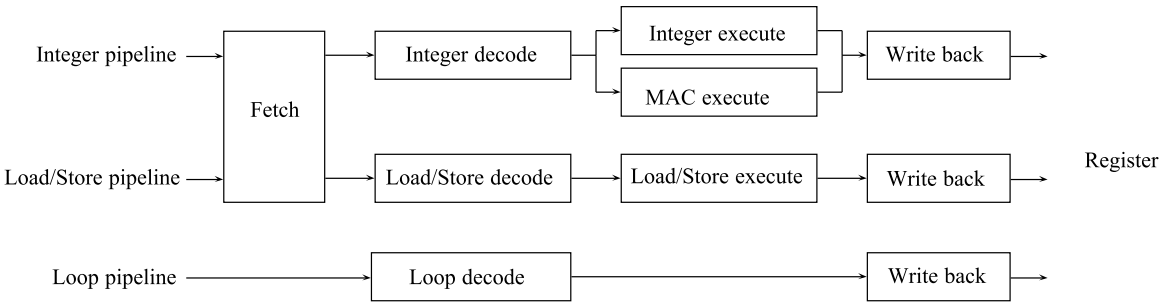

Figure 5.3: The three parallel pipelines of the execution unit

registers can be combined to one 64-bit register E.

5.3 TriCore Instruction Set

The TriCore supports 16 and 32-bit instructions. Since there is no mode bit and no requirement for word alignment of the 32-bit instructions we can freely mix the instructions. To specify a 16-bit command, the basic operation name is followed by a 16, e. g., `SH16`. The advantage of a 16-bit instruction is, that it is smaller than a 32-bit instruction in the program memory. It should be used whenever possible for smaller code size.

To follow the upcoming assembler examples, we first show how an assembler instruction is composed. The assembler instruction starts with the *basic operation*, which is derived from the intention of the instruction. For example, a shift instruction starts with `SH`. The basic operation can be followed by an *operation modifier*, e. g., the instruction `JNE` would execute a conditional jump. The condition here is given by the `NE`, which stands for "not equal". In Table 5.1 we list all operation modifiers used by us.

Operation Modifier	Name	Description	Example
C	Carry	Use and update carry bit	ADDC
S	Saturation	Saturate result	ADDS
EQ	Equal	Comparison equal	JEQ
GE	Greater than	Comparison greater than or equal	JGE
A	Absolute	Absolute (jump)	JLA
I	Immediate	Large immediate	ADDI
LT	Less than	Comparison less than	JLI
NE	Not equal	Comparison not equal	JNE
D	Decrement	Decrement counter	JNED
I	Increment	Increment counter	JNEI
Z	Zero	Use zero immediate	JNZ

Table 5.1: Operation Modifiers

The *data type modifier* specifies the data type we want to work with. For example, if we load a byte from memory, we use the `LD.B` instruction. The `.B` denotes, that we load a byte value. In Table 5.2 we list the modifier for the individual data types:

The data type modifiers can be combined to a new one, e. g., the instruction `LD.BU` loads an unsigned byte. In the following subsections, we will describe the commonly used assembler instructions.

5.3.1 Load and Store Instructions

To load and store a value into data register `D[A]` we can use the load word and store word instructions (`LD.W` and `ST.W`).

```
LD.W            D[a], offset            ; Load Word (Absolute
                                        ;   Addressing)
```

Data Type Modifier	Name	Example
.D	Double-word	LD.D
.W	Word	EQ.W
.A	Address	ADD.A
.H	Half-Word	ADD.H
.B	Byte	ADD.B
.T	Bit	AND.T
.U	Unsigned	ADDS.U

Table 5.2: Data Type Modifiers

```
LD.W            D[a], A[b], offset      ; Load Word (Base + Short
                                        ;   Offset Addressing)

ST.W            off, D[a]               ; Store Word
```

5.3.2 Arithmetic Instructions

The following three bit arithmetic instruction work almost the same way. So we describe it only once. The `OR,XOR` and `AND` instruction computes the bitwise OR, XOR, respectively AND operation of the data register `D[a]` and the data register `D[b]`. The result is stored in the data register `D[c]`.

```
OR              D[c], D[a], D[b]        ; c = a OR b
XOR             D[c], D[a], D[b]        ; c = a XOR b
AND             D[c], D[a], D[b]        ; c = a AND b
```

The multiplication instruction multiplies two signed 32-bit values from the data register `D[a]` and the data register `D[b]` and puts the product into the 32-bit data register `D[c]` or the 64-bit data register `E[c]`.

```
MUL             D[c], D[a], D[b]        ; c = (lower half of)
                                        ; a * b

MUL             E[c], D[a], D[b]        ; c = a * b
```

With the shift instruction `SH` we can shift the value in `D[a]` by the number of bytes specified with either a constant value `const` or the value in data register `D[b]`. The result is put in `D[c]`. The vacated bits are filled with zeros and the bits shifted out are discarded.

```
SH              D[c], D[a], D[b]        ; shift D[a] by D[b] positions
                                        ; and store the result in D[c]

SH              D[c], D[a], const       ; shift D[a] by #const positions
                                        ; and store the result in D[c]
```

5.3.3 Extract Instruction

It is possible to work with so called bit-fields. We can extract the number of consecutive bits specified by `width` starting at the bit `pos` from a source register `D[a]` with the `EXTR` (Extract Bit Field) and `EXTR.U` (Extract Bit Field Unsigned) instructions, beginning with the bit number specified by the `pos` operand. The result is stored sign extended (`EXTR`) or filled with zeros (`EXTR.U`) in the destination register `D[c]`. Figure 5.4 denotes the `EXTR.U` operation.

```
EXTR            D[c], D[a], pos, width  ; Extract Bit Field
EXTR.U          D[c], D[a], pos, width  ; Extract Bit Field Unsigned
```

To extract a 32-bit word from the registers {`D[a]` and `D[b]`}, where `D[a]` contains the most-significant 32 bits of the value, we use the `DEXTR` instruction. The extraction starts at the bit number specified with `32-pos`. The result is put in `D[C]`.

```
DEXTR           D[c], D[a], D[b], pos   ; Extract from Double Register
```

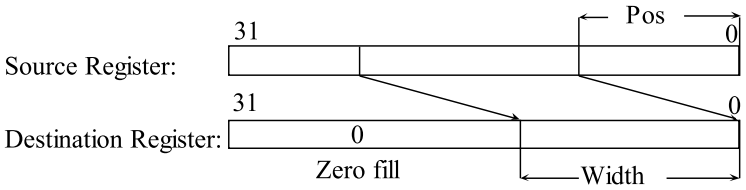

Figure 5.4: Operation of the EXTR.U instruction, source:[Inf07]

Packed Arithmetic

The packed arithmetic instruction partitions a 32-bit word into four byte values or into two, 16-bit halfwords values. Those can be fetched, stored, and operated on in parallel. The packed byte format is denoted in Figure 5.5. Instructions which operate on the data in this way are denoted by the `.B` and `.BU` data type modifier. The arithmetic on packed data includes addition, subtraction, multiplication, shift and the absolute difference.

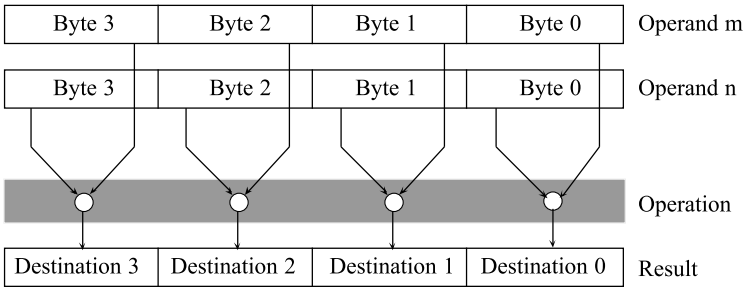

Figure 5.5: Packed Byte Data Format, source:[Inf07]

To load the contents of the memory location specified by the `offset` we can use the load byte instruction (`LD.B`).

```
LD.B            D[a], offset            ; Load Byte (Absolute
                                        ;   Addressing)

LD.B            D[a], A[b], offset      ; Load Byte (Base + Short
                                        ;   Offset Addressing)
```

The `ST.B` instruction stores the byte value in the eight least-significant bits of the data register `D[a]` to the byte memory location specified by the offset `off`.

```
ST.B off, D[a] ; Store Byte
```

5.3.4 Address Arithmetic

The `ADDSC.A` operation left-shifts the contents of data register `D[a]` by the amount specified by `n`, where n = 0,...,3. That value is added to the contents of address register `A[b]` and the result is put into address register `A[c]`.

```
ADDSC.A         A[c], A[a], D[a], n     ; Add Scaled Index to
                                        ; Address
```

The `LEA` instruction computes the effective address and puts the result in address register `A[a]`.

```
LEA             A[a], offset            ; Absolute Addressing
                                        ; Mode

LEA             A[a], A[b], offset      ; Base + Short Offset
                                        ; Addressing Mode
```

5.4 Time Measurement

The TriCore has a system timer (STM), which we can utilize for our time measurement, see the user's manual [Inf07, Section 15]. The STM is a free running 56-bit upward counter and driven by max. 75 MHz (=f_{SYS}). The maximum clock period is $2^{56} \cdot f_{STM}$. For f_{STM}=75 MHz, for example, the STM counts 30.46 years before it overflows. The minimum resolution of the STM is 13.3 ns. The default frequency after a reset is $f_{\mathrm{STM}} = f_{\mathrm{SYS}}/2$.

We can read the counter value from the STM timer register, which has the address $\{\texttt{F0000210}\}_{16}$. Due to the fact, that the STM is 56-bit wide, we cannot read its entire content with one instruction. We have to use two load instructions. Listing 5.1 denotes, how we can load the STM register values with inline assembler.

The STM value can also be loaded in sections from seven registers, STM_TIM0 through STM_TIM6. Theses register can be viewed as individual 32-bit timers, each with a different resolution and timing range [Inf07, Section 15].

```
1 inline tcclock_t tcclock( void ) {
2   uint64 tmp;
3
```

```
4   /* Load 56-bit system timer value */
5   __asm(
6     "LD.W %0.0, 0xf0000210"
7     "LD.W %0.1, 0xf000022c"
8       : "=e" (tmp)
9   );
10
11  return tmp;
12 }
```

Listing 5.1: The function tcclock returns the current system timer STM value. Source: Timing measurement routines from Robert Szerwinski.

The `"=e" (tmp)` in line 8 denotes, that the variable `tmp` is used for the result of the `LD.W` instruction and is an extended 64-bit data register.

Chapter 6

Optimized and Randomized AES Implementations on TriCore

6.1 Implementation Constraints

The AES implementation runs on an automotive processor and it will not be the only task running there, so it will be implemented as space saving as possible in terms of RAM and ROM consumption[1].

The ROM consumption is measured for a whole minimal AES implementation. We get the ROM consumption from the Assembler list file. To generate the list file, we have to enable the "Generate list file" option in the project settings dialog "Assembler -> List file" from the Tasking toolchain. The generated list file contains a "List section summary" with the ROM consumption.

6.1.1 Validation Tests

The National Institute of Standards and Technology (NIST) requires that all AES implementations have to pass some algorithm validation tests. This tests help to find failures early during the development process. It can happen, e. g., that we calculate a wrong byte in a S-box which is not used during an encryption but during a later one. This can happen because the used S-box bytes depend on the used plaintext and key combinations. To verify the correctness of the implementation, the following four tests are used:

- Variable key known answer test,
- Variable text (plaintext/ciphertext) known answer test, and
- tables known answer test,
- Monte Carlo test.

Variable Key Known Answer Test

During the Key Known Answer Test (Key KAT) [NIS98], the 128-bit plaintext is always initialized to zero. The key used in an AES iteration i, $0 \leq i < 127$ consists of a one in the i^{th} position and zeros in all other positions. Each of the possible keys is created by shifting the one a single position at a time, starting at the most significant (left most) bit of the key.

1 The requirements are very different depending on the application. But, in general, speed is not the most critical requirement, whereas memory and program space footprint are almost always an issue.

Variable Text (Plaintext/Ciphertext) Known Answer Test

In this test, the 128-bit key is initialized to zero. Each 128-bit block data input consists of a one at the i^{th} position, starting at the most significant (left most) bit position, where $i = 0, \ldots, 127$ is the AES iteration.

Table Known Answer Test

This test contain sets of key and plaintext combinations which have been carefully selected to test the tables used for the AES implementation, e. g., the S-boxes.

Monte Carlo Test

This test is used to identify implementation flaws. Therefore the authors of AES supply pseudo random values for the key and plaintext. The test consists of 4 million AES encryption/decryption cycles. Thereby the results of every $10,000^{th}$ encryption or decryption cycle are recorded and evaluated with the supplied results from the authors. The first key k_0 and plaintext p_0 are given. The following keys and plaintexts are taken from the encryption like it is depicted in Algorithm 7.

Algorithm 7: Monte Carlo Test – ECB Encryption

Input: k_0, p_0
for $i = 0$ **to** 399 **do**
    for $j = 0$ **to** 9999 **do**
        c_j =aes_Encrypt(k_i, p_j)
        $p_{j+1} = c_j$
    end
    $k_{i+1} = k_i \oplus c_j$
end

6.2 Available AES Implementations

We begin our work by reviewing the already existing AES implementations from Brijesh Singh [Sin08] and Robert Szerwinski, done at Robert Bosch GmbH, Corporate Research[2].

The implementations are 8-bit implementations and work on the 16-byte State matrix, as it is described in the AES overview, Section 2.1. The State matrix is filled sequentially with `memcpy()` which is a few cycles faster than copying the plaintext byte-by-byte within a `for()` loop. Both authors combine the *SubBytes* and the *ShiftRows* transformation. This new transformation realizes the *SubBytes* transformation as a table lookup and integrates the *ShiftRows* transformation by using the indices for the table lookups, as they were after a *ShiftRows* transformation. This trick safes the rotation operations of the *ShiftRows* transformation, as described in Section 2.3.4.

Now we come to the differences of the available implementations. Brijesh Singh uses separate key scheduling in his implementation. He realized the *xtimes* operation as a table lookup.

2 Note that all versions are in a very early release state and are not production ready!

This idea comes from Tom St Denis [Den07]. For this lookup table he needs 256 bytes additional ROM. In the following we refer to this implementation as *Singh-1*. The second implementation of Brijesh Singh also uses a separate key scheduling but computes the *xtimes* operation as it is described in Section 2.3. We refer to this second implementation as *Singh-2*.

Robert Szerwinski realized the AES implementation without the lookup table for the *xtimes* operation. He implements this function as it was suggested in the standard and is described in the mathematical background, Section 2.2. Contrary to the standard, he calculates the keys on-the-fly. We refer to his implementation as *Szerwinski*.

6.3 General Optimizations Hints

Both presented implementations neither uses the 32-bit register size nor any special instruction of the TriCore. But they are a good start for our further optimization.

Since we program our implementation in `C`, we have to pay attention to the general `C` compiler specific behavior. For example we pass constant variables to functions as `const` values, which avoid the compiler from creating a local copy of the variable in the called function. Brijesh Singh also notes that using the `__near` pragma[3] before a variable declaration causes the linker to put the variable into direct addressable memory, which makes the access to the variable faster. Another known fact is, that calling functions is very expensive. We avoid functions if they are used often, for example in loops. Alternatively for those functions we create a macro or an `inline` function of it. The compiler then replaces the macro resp. `inline` function call by the actual code of the function.

6.4 AES Optimizations for the TriCore

In the following sections we will walk through the four AES transformations *AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns* and take a look, if they can be optimized for the TriCore. In addition, we will use the 32-bit capability of the TriCore and implement the transformations as discussed in Section 2.3.5.

6.4.1 AddRoundKey

The *AddRoundKey* transformation is implemented as macro to avoid the overhead of a function call. Because we arrange the State matrix and the expanded key arranged column wise on RAM, we can load one word `w` from the State matrix and one word from the expanded key in two clock cycles.

Listing 6.1 denotes how a column of the expanded round key `expkey` is added to one column `w` of the State matrix. The address for the used key is calculated from the current `round` and the current `column` index of the State matrix.

```
1 #define AddRoundKeyColumn(w, expkey, round, column) \
2     w = w ^ expkey[(4U * round) + column];
```

Listing 6.1: AddRoundKey transformation for one column of the State matrix

3 In computer science or software engineering, a pragma is a compiler directive communicating additional "pragmatic" or implementation-specific information (Wikipedia).

The compiler creates the following Assembler instructions for the 32-bit addition of one column from the State matrix:

```
LD16.W          D0,    [A10]            ; load word from the
                                        ; State matrix

LD16.W          D15,   [A15]16          ; load key word from expanded
                                        ; key

XOR             D0,    D15              ; add them

ST16.W          [A10], D0               ; write the word back to RAM
```

The `LD.W` (load word) instruction, loads 32 subsequent bits from the State matrix, which is located in RAM, into the data register `D[0]`. The next instruction loads 32-bits from the expanded key to `D[15]`. Both values are bitwise added with the `XOR` instruction which the result in `D[0]`. The `ST.W` (store word) instruction stores the register value at the given address in `A[10]` (the position of the State matrix in RAM).

By processing four bytes at the same time, we save approximately the factor four for the key addition, i. e., the AddRoundKey takes 176 cc instead of 704 cc. An *AddRoundKey* transformation on the whole State matrix is denoted in Listing 6.2:

```
1 #define AddRoundKey(state,expkey,round)\
2   AddRoundKeyColumn (state[0], expkey, round, 0);\
3   AddRoundKeyColumn (state[1], expkey, round, 1);\
4   AddRoundKeyColumn (state[2], expkey, round, 2);\
5   AddRoundKeyColumn (state[3], expkey, round, 3);
```

Listing 6.2: Listing of the 32-bit AddRoundKey transformation

6.4.2 SubBytes and ShiftRows

We also combine the *SubBytes* transformation with the *ShiftRows* transformation by a priori choosing the right index value for the table lookup.

During the *SubBytes* calculation we have to access each byte $s_{r,c}$ of the State matrix for processing the table lookup (the S-box). To access each byte of the State matrix instead of the whole column, we have to cast the 32-bit value from the State matrix to an 8-bit value as it is done in Listing 6.3.

```
1 static void SubBytesShiftRows(word state)
2   begin
3     byte tmp;
4     byte s = (byte) state;
5
6     /* column one */
7     s[0] = sbox[s[0]];
8     s[4] = sbox[s[4]];
```

```
9       s[8] = sbox[s[8]];
10      s[12] = sbox[s[12]];
11
12      /* ... Process the columns two, three and four ... */
13    end
```

Listing 6.3: Listing of the combined SybBytes and ShiftRows transformation

This causes the compiler to use the byte data type modifiers, e. g., `LD.BU` (load byte unsigned). The `LEA` instruction computes the absolute address of the S-box and puts the resulting address in address register `A[4]`. The `ADDSC.A` (add scaled index to address) instruction calculates the index value and puts the resulting address in address register `A[2]`. Finally, the S-box byte at the calculated index is stored in the State matrix. The following assembler listing depicts the combined *SubBytesShiftRows* operation for one byte of the State matrix. Address register `A[15]` contains the next byte which is used as index for the TLU in the S-box (line three).

```
LEA             A4, sbox                ; compute absolute address
                                        ; of the S-box

LD16.BU         D15, [A15]              ; load byte from the State
                                        ; matrix

ADDSC16.A       A2, A4,D15,#0           ; use the byte as index
                                        ; for the TLU

LD16.BU         D15, [A2]               ; load the TLU value

ST16.BU         [A15], D15              ; store the looked up value in
                                        ; the State matrix
```

We have to do these steps (excluding the `LEA` operation) for all 16 bytes bytes of the State matrix. Since the State matrix of the reference implementations are also processed byte-wise, we cannot achieve an speed improvement here.

6.4.3 MixColumns

The *MixColumns* transformation is well discussed in Section 2.3.5. We start with implementing the multiplication function `FFmulX` in Listing 6.4.

Instead of the shift instructions which are used by Gladman, we use the multiplication by $\{\texttt{1B}\}_{16}$ for the reduction if an overflow occurs. We do this because the TriCore can multiply in two clock cycles, which is still faster than the alternative shift and XOR method which is used by Gladman and is denoted in Listing 2.5.

```
1 /* multiply the four bytes of w in parallel and reduce the result
2  * with 0x1B (if the highest bit of the byte is set)
3  */
4 word FFmulX(word w)
```

```
5   begin
6     word t = w & 0x80808080;
7
8     return ((w ^ t) << 1) ^ ((t>>7) * 0x0000001B);
9   end
10 }
```

Listing 6.4: Parallel multiplication of the four bytes in word `w` by two

To implement the `MixColumn` operation from Listing 2.6 efficiently, we need the ability to cyclically rotate the content of a 32-bit register by one, two and three bytes. If we take a look at the instruction manual of the TriCore [Inf08b], we cannot find a direct instruction for a rotation. Fortunately, we can utilize the `DEXTR` (extract from double register) instruction for the rotation. The function `__my_rot1` in Listing 6.5 returns the word `w` rotated by one byte position to the left. To rotate the word `w` by two and three bytes the same instruction can be used with a different starting bit, i. e., 16 for a rotation by two, and 24 for a rotation by three bytes.

```
1 word ___my_rot1(word w)
2   begin
3     word tmp;
4
5     /* Rotate the content of word w one byte */
6     ___asm("dextr %0, %1, %1, #8": "=d"(tmp):"d"(w));
7
8     return tmp;
9   end
```

Listing 6.5: Rotate the four bytes of `w` one byte to the left

Listing 6.6 denotes the 32-bit implementation of the *MixColumns* transformation. The function `MixColumn` calculates the MixColumns operation for the column `w` with four rotations and four XOR operations. The `MixColumns` function in line six applies this transformation on each column of the State matrix.

```
1 word MixColumn (word w)
2   begin
3     w = ___my_rot3(w) ^ ___my_rot2(w) ^
4           ___my_rot1(w) ^ FFmulX(w ^ ___my_rot3(w));
5     return w;
6   end
7
8  MixColumns(word s[])
9   begin
10    MixColumn(state[0]);
11    MixColumn (state[1]);
12    MixColumn (state[2]);
13    MixColumn (state[3]);
```

```
14   end
```

Listing 6.6: 32-bit version of MixColumns

6.4.4 Inverse MixColumns

Listing 6.7 depicts the *InvMixColumns* transformation. The *InvMixColumn* operation is applied on each column `w` of the State matrix. The implementation follows the transformation description in Section 2.3.5.

```
1  word InvMixColumn (word w)
2    begin
3      word tmp = 0;
4
5      tmp = ___my_rot2 (w);
6      tmp = tmp ^ w;
7      tmp = FFmulX (tmp);
8      tmp = FFmulX (tmp);
9
10     w = w ^ tmp;
11
12     w = MixColumn(w);
13
14     return w;
15   end
16
17 InvMixColumns (word state[])
18   begin
19     InvMixColumn (&state[0]);
20     InvMixColumn (&state[1]);
21     InvMixColumn (&state[2]);
22     InvMixColumn (&state[3]);
23   end
```

Listing 6.7: InvMixColumns transformation with 32-bit operations

6.5 Size Comparison of Unprotected Implementations

With the above discussed additional optimizations, we can achieve the memory consumption presented in Table 6.1. The values are taken from the assembler listing file, which depicts the data and code section size. The data section from Singh-1 contains the 256-byte *xtimes* table, the 12 bytes round constant `RC` and the 256 bytes S-box. This leads to a total data memory consumption of 524 bytes.

The data memory of Singh-2 does not contain the 256-byte *xtimes* table, it consists of the S-box 256 bytes and 12 bytes for the round constant `RC`. The Szerwinski implementation only holds the 256-byte S-box in the data memory.

Implementation	**Code** (bytes)	**Data** (bytes)
Singh-1	720	524
Singh-2	724	268
Szerwinski	612	256
Optimized AES	1494	528

Table 6.1: Memory consumption comparison between the AES reference implementation from Brijesh Singh and the Optimized TriCore implementation

It is obviously that the program code of the Optimized AES implementation is about twice as large as the other implementations. This is because it includes encryption and decryption whereas the first three implementations only include the encryption. The data memory contains the S-box, the inverse S-box for the decryption, the 12-byte round constant `RC` which is needed by the key expansion (Section 2.3.2) and four byte for the constant word $\{80808080\}_{16}$ which is needed in the `FFmulX` function. The performance analysis will be done in Section 7.

6.6 Protected AES Implementation

The protected AES implementation uses masking, which is described in Section 4.2 and randomization from Section 4.3 to be resistant against timing and first order DPA attacks. We add D additional dummy operations to the real transformations. From these fixed D dummy operations, a random number of $D1$ dummy operations, $D1 \leq D$ are executed before the first round begins. After the last round, $D - D1$ dummy operations are executed. The dummy operations work one a dummy State matrix DS with a dummy key DK.

Figure 6.1 denotes the program flow of the protected AES implementation. In the randomization zones, the transformations are shuffled. This means for the *AddRoundKey* and *MixColumns* transformations, that the starting column of the State matrix for these transformations is randomly chosen with two random bits rc. During the *SubBytes* transformation, the first column is randomly chosen as well as the first row. The random starting row is selected with the two random bits rr.

The currently used masks is denoted on the right side in Figure 6.1. The four mask bytes M_1 to M_4 are used to mask the *MixColumns* transformation, as it is described in Section 4.2.5. The *SubBytes* transformation is masked with the random mask M.

For masking and randomization 320 random bits are needed per AES encryption. Table 6.2 summarizes the names and usage of the random bits.

6.6.1 Masking the Optimized AES Implementation

We have outlined two critical operations which must be protected against a differential power analysis attack. The initial two *SubBytes* transformations during the first two rounds as well as the last two *SubBytes* transformations. We will present two independent masking schemes. To protect the implementations against first order DPA attacks, we will use additive masking from Section 4.2.1. In addition we will implement the Combined Masking in Tower Fields

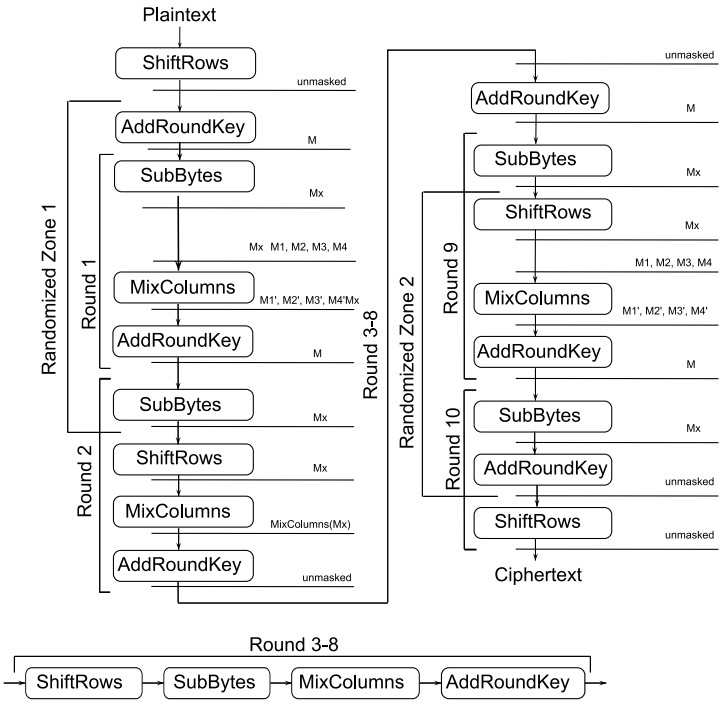

Figure 6.1: Protected AES

Name	No. of random bits	Usage
D1	8	No. of dummy operations before the real cipher begins
rc	4	Random entry column
rr	4	Random entry row
Mx	8	Masks the S-box entries
M	8	Masks the S-box entry indices
M_1	8	Masks the first column during a MixColumns transf.
M_2	8	Masks the second column during a MixColumns transf.
M_3	8	Masks the third column during a MixColumns transf.
M_4	8	Masks the forth column during a MixColumns transf.
DS	128	Dummy State matrix
DK	128	Dummy Key

Table 6.2: Number of random values required for the protected AES Implementation wit additive masking

approach from Section 4.2.4. Both masking schemes can be exchanged in the protected AES implementation.

Additively Masked SubBytes

To mask the *SubBytes* transformation additively, the whole S-box has to be recalculated, see Section 4.2.1. Because the recomputing of the S-box is an expensive operation in terms of runtime, we make it configurable. We can decide during the compilation of the implementation, after how many AES encryptions the S-box will be recalculated.

Listing 6.8 shows, how the new masked S-box (`maskedSBox`) is calculated. The two random values `M` and `Mx` mask the new created S-box.

```
1 static inline void aes_MaskSBox ( uint8 * maskedSBox,
2         const uint8 sbox[], const uint8 M, const uint8 Mx) {
3
4     int i;
5
6     for (i = 0; i < 256; i++) {
7         maskedSBox[i ^ M] = sbox[i] ^ Mx;
8     }
9 }
```

Listing 6.8: Function to calculate the masked S-box

To perform a *SubBytes* transformation with the masked S-box, we have to add (XOR) the value M to every byte $s_{r,c}$ of the current State matrix. After the masked transformation, we have to remove the mask Mx from the looked-up value.

Two possibilities are conceivable. The mask values M and Mx can be added in the key scheduling and mask the expanded round. Then the current round key would mask the State in such a way we need it. But that implies, that we have to calculate the expanded key every time, we change the masks for the S-box. Since our implementation is targeted on devices with a fixed key, a recalculation of the expanded key would be very wasteful in terms of runtime.

We chose the *AddRoundKey* transformation, to add the currently needed mask values to the State. Listing 6.9 denotes the new `AddRoundKeyColumn` function, which additionally adds the mask value `m` to the column `w` of the current State matrix.

```
1 static inline void
2 AddRoundKeyColumn (uint32 * w, const uint32 expkey[], const uint8
3     round, const uint32 m, const uint8 column) {
4   *w ^= (expkey[(4U * round) + column] ^ m);
5 }
```

Listing 6.9: Function to add the mask m and the current round key to the word w of the current State matrix

SubBytes Masked with the CMTF Approach

As we have discussed in Section 2.3.8, we can calculate the inversion in so called composite fields. In this section we show how the equations from Section 4.2.4 can be computed. This can also be found in [WOL02]. First we have to transform an element $a \in \mathrm{GF}(2^8)$ to its isomorphic composite field representation $a_h x + a_l, \quad a_h, a_l \in \mathrm{GF}(2^4)$:

$$a \cong a_h x + a_l, \quad a \in GF(2^8), \quad a_h, a_l \in GF(2^4). \tag{6.1}$$

The transformation between the field representations can be calculated the following way:

$$\begin{aligned}
a_A &= a_1 \oplus a_7, & a_B &= a_5 \oplus a_7,\\
a_C &= a_4 \oplus a_6, & &\\
a_{l0} &= a_C \oplus a_0 \oplus a_5, & a_{l1} &= a_1 \oplus a_2,\\
a_{l2} &= a_A, & a_{l3} &= a_2 \oplus a_4,\\
a_{h0} &= a_C \oplus a_5, & a_{h1} &= a_A \oplus a_C,\\
a_{h2} &= a_B \oplus a_2 \oplus a_3, & a_{h3} &= a_B,
\end{aligned}$$

where $a_i, i = 0, \ldots, 7$ are the bits from $a \in \mathrm{GF}(2^8)$ and $a_{hj}, a_{lj}, j = 0, \ldots, 3$ are the bits from $a_h, a_l \in \mathrm{GF}(2^4)$. The $\oplus$ denotes the XOR operation in GF(2).

The inverse transformation, which converts the two-term polynomial $a_h x + a_l$ back into a $\mathrm{GF}(2^8)$ element $a \in GF(2^8)$ can be calculated following way:

$$\begin{aligned}
a_A &= a_{l1} \oplus a_{h3}, & a_B &= a_{h0} \oplus a_{h1},\\
a_0 &= a_{l0} \oplus a_{h0}, & a_1 &= a_B \oplus a_{h3},\\
a_2 &= a_A \oplus a_B, & a_3 &= a_B \oplus a_{l1} \oplus a_{h2},\\
a_4 &= a_A \oplus a_B \oplus a_{l3}, & a_5 &= a_B \oplus a_{l2},\\
a_6 &= a_A \oplus a_{l2} \oplus a_{l3} \oplus a_{h0}, & a_7 &= a_B \oplus a_{l2} \oplus a_{h3}.
\end{aligned}$$

To compute the inversion in the composite field $\mathrm{GF}((2^4)^2)$ an addition, multiplication, and inversion is needed. Two elements from $\mathrm{GF}(2^4)$ can be added the following way:

$$(a_h x + a_l) \oplus (b_h x + b_l) = (a_h \oplus b_h)x + (a_l \oplus b_l). \tag{6.2}$$

The multiplication of two elements, which is defined in Section 2.3.8:

$$q(x) = a(x) \cdot b(x) \bmod m_4(x), \quad a, b, q \in GF(2^4) \tag{6.3}$$

can be computed the following way:

$$\begin{aligned}
a_A &= a_0 \oplus a_3, & a_B &= a_2 \oplus a_3,\\
q_0 &= a_0 b_0 \oplus a_3 b_1 \oplus a_2 b_2 \oplus a_1 b_3, & q_1 &= a_1 b_0 \oplus a_A b_1 \oplus a_B b_2 \oplus (a_1 \oplus a_2) b_3,\\
q_2 &= a_2 b_0 \oplus a_1 b_1 \oplus a_A b_2 \oplus a_B b_3, & q_3 &= a_3 b_0 \oplus a_2 b_1 \oplus a_1 b_2 \oplus a_A b_3,
\end{aligned}$$

where $a_i, b_i, q_i, i = 0, \ldots, 3$ are the bits from a, b and $q \in \mathrm{GF}(2^4)$.

To square an element in GF(2^4):

$$q(x) = a(x)^2 \bmod m_4(x) \tag{6.4}$$

we calculate:

$$\begin{aligned} q_0 &= a_0 \oplus a_2, & q_1 &= a_2, \\ q_2 &= a_1 \oplus a_3, & q_3 &= a_3. \end{aligned}$$

The multiplicative inversion a^{-1} of an element $a \in \mathrm{GF}(2^4)$ is calculated by the following formula:

$$\begin{aligned} a_A &= a_1 \oplus a_2 \oplus a_3 \oplus a_1a_2a_3 \\ q_0 &= a_A \oplus a_0 \oplus a_0a_2 \oplus a_1a_2 \oplus a_0a_1a_2 & q_1 &= a_0a_1 \oplus a_0a_2 \oplus a_1a_2 \oplus a_3 \oplus a_1a_3 \oplus a_0a_1a_3 \\ q_2 &= a_0a_1 \oplus a_2 \oplus a_0a_2 \oplus a_3 \oplus a_0a_3 \oplus a_0a_2a_3 & q_3 &= a_A \oplus a_0a_3 \oplus a_1a_3 \oplus a_2a_3 \end{aligned}$$

Now we can calculate the inversion of a whole two-term polynomial. Note that, this time, the addition $\oplus$ and multiplication $\otimes$ are in GF(2^4) and are computed as we have discussed above. The inversion of the whole two-term polynomial is done by calculating:

$$(a_hx + a_l)^{-1} = a'_hx + a'_l = (a_h \otimes d)x + (a_h \oplus a_l) \otimes d \tag{6.5}$$

where:

$$d = ((a_h^2 \otimes \{E\}_{16}) \oplus (a_h \otimes a_l) \oplus a_l^2)^{-1}, \quad d \in \mathrm{GF}(2^4) \tag{6.6}$$

As we can see[4], the transformation between the two field representations and the computation of the inverse are very expensive computations because they work bitwise on the elements. This values therefore will be pre-calculated stored in four lookup tables T_{d_1}, T_{d_2}, T_m and T_{inv} as it is discussed in Section 4.2.4. Listing 6.10 denotes the inversion with these four tables. This function gets a byte `x` from the State matrix and the mask `m` which masks the calculation. The functions `mapGF16` and `imapGF16` transform the elements between the field representations.

```
1  byte maskedInvertGF256(byte x, byte m)
2    begin
3      byte xh, xl, mh, ml;
4      byte d, di;
5      byte ah, al, a;
6
7      mapGF16(x, &xh, &xl);
8      mapGF16(m, &mh, &ml);
9
10     d = td1[xh][mh] ^ td2[xh][xl] ^ tm[xh][ml] ^
11                                tm[xl][mh] ^ tm[mh^ml][ml];
12
13     di = tinv[d][mh];
```

4 by counting the XOR and AND operations, where the concatenation of two bits a_ia_j represents an AND

```
14
15      ah = tm[xh][di^mh^ml] ^ mh ^ tm[di^mh^ml][mh] ^
16                                tm[xh][ml]^tm[mh][ml];
17
18      al = tm[xl][di]^ml^tm[di][ml] ^ tm[xl][mh] ^
19                                ah ^ mh ^ tm[mh][ml];
20
21      imapGF16(&a, ah, al);
22
23      return a;
24    end
```

Listing 6.10: Function that calculates the inversion part of the SubBytes transformation with the CMTF approach

Listing 6.11 depicts the calculation of Equation (2.12). The function `rot1_right` rotates the byte `b` bitwise to the right. The function `rot1_left` rotates the byte `b` bitwise to the left. The function `parity` computes the parity bit of the given byte.

```
1  byte aff_trans(byte a, byte c)
2    begin
3      byte t = 0;
4      byte b = 0;
5      byte mask = 0xF8;
6
7      for (int i=0; i<=7; i++) {
8        t = a & mask;
9
10       mask = rot1_right(mask);
11       b = rot1_left(b);
12       b = b ^ parity(t);
13     }
14
15     b = b ^ c;
16
17     return b;
18   end
```

Listing 6.11: Function to calculate the affine transformation part of the SubBytes transformation

The whole masked *SubBytes* transformation for one byte of the State matrix is denoted in Listing 6.12.

```
1  byte maskedSubBytes(byte x, byte m)
2    begin
3      /*
4       * Input:    Byte from the State matrix: x, mask: m
5       * Output:   inv=SubBytes(a) XOR m1 where m1 = aff_trans(m)
```

```
6       */
7
8       byte inv = maskedInvertGF256(x, m);
9       aff_trans(&inv, 0x63);
10
11      return inv;
12    end
```

Listing 6.12: Function to calculate the SubBytes transformation with the CMTF approach

6.6.2 Size Comparison of Protected Implementations

With the above discussed protected implementations, we can achieve the memory consumption presented in Table 6.3. Both AES implementations are protected with dummy operations and shuffling. The first uses additive masking to mask as masking aproach, the second uses CMTF as masking aproach. Both implementations contain the code for encryption and decryption.

Implementation	**Code** (bytes)	**Data** (bytes)
protected with additive masking	4480	1046
protected with CMTF	4772	1552

Table 6.3: Memory consumption comparison between the protected AES implementation

The code size increases for the additive masked implementation, because we have additional functions that work on columns instead on the whole State matrix for the individual transformations.

Table 6.4 denotes the content of the data section for the protected AES version with additive masking, which is in total 1046 bytes.

Name	**No. of bytes**	**Usage**
const	4	Constant for the xtimes function
`RC`	12	Round constant
S-box	256	S-box for encryption
IS-box	256	S-box for decryption
masked-Sbox	256	Masks S-box for encryption
masked-ISbox	256	Masks S-box for decryption
$2 \cdot$ M	2	Masks the S-box entries (enc./dec.)
$2 \cdot$ MX	2	Masks the S-box indices (enc./dec.)
$2 \cdot$ MaskRoundCounter	2	Use counter for the masked S-boxes

Table 6.4: Content of the data section for the protected AES Implementation with additive masking

Table 6.5 denotes the content of the data section for the protected CMTF AES version. We need 1552 bytes data memory for this implementation.

Name	**No. of bytes**	**Usage**
const	4	Constant for the xtimes function
RC	12	Round constant
tm	256	Table TM
tinv	256	Table TINV
td1	256	Table TD1
td2	256	Table TD2
map	256	Table to map an element from GF(2^8) to GF(2^4)
imap	256	Table to map an element from GF(2^4) to GF(2^8)

Table 6.5: Content of the data section for the protected AES Implementation with CMTF

Chapter 7

Implementation Results

In this chapter, the measurement setup is discussed. The runtime of the unprotected implementations is measured and compared in the following. The runtime behavior of the protected AES implementations is analyzed in detail.

7.1 Measurement Setup

To compare the four unprotected AES implementations in size and speed, the given source files are imported to the Tasking development environment. Therefore an own project for every implementation is created where the following default build settings have been changed:

- optimization: *level 2 (Optimize more)*,
- trade-off between speed and size: *Level 4 (Size)*, and
- active build configuration: *Release*.

To measure the timing behavior of the AES implementations, the TriCore 1796 emulation device (TC1796ED) development board is used, see Section 5.1.4.

The statistical analysis were carried out on a PC featuring a Intel Celeron processor at 2.53 GHz and 960 MB RAM. We used Matlab Release 2007b with the Statistics Toolbox version 6.1 [Mat07].

7.2 Runtime Comparison of the Implementations

For the runtime measurement, we use the maximal CPU frequency of 150 MHz. The runtime $X_j, j = 1, \ldots, 1000$ of `aes_Encrypt` is measured for $1,000$ different encryptions. Thereby we use one fixed key k and different plaintexts p. The new plaintext p_{j+1} we use is the output c_j of the previous encryption $c_j = \texttt{aes\_Encrypt}(k, p_j)$. Algorithm 8 shows the program fragment used for the measurement process. We start the measurement right before the AES run which is denoted by the `tic` instruction. After the run we stop the measurement (denoted by `toc`) and save the elapsed runtime. The runtime X can be seen as a random variable that is normally distributed, i. e. $X = x_1, \ldots, x_{1000}$ with $X_j \sim N(\mu_x, \sigma_x^2), x_j$ iid. We will compare the mean value $\overline{X}$ of X of the given implementations in Table 7.1.

The optimized AES implementation is with 78.31 μs (204.32 kbyte/s) the fastest implementation. It is about 2.6 times faster than the comparable implementation from Singh-2, which also uses an expanded key and only table lookups for the *SubBytes* transformation. The faster

Algorithm 8: Pseudo code of the runtime measurement process. The runtime of aes_Encrypt is measured and stored in X_j for every execution.

Input: $k = \{\texttt{000102030506070809A0B0C0D0F101112}\}_{16}$, p_1
for $j = 1$ **to** 1000 **do**
  tic;
  c_j = aes_Encrypt(k, p_j);
  X_j = toc;
  $p_{j+1} = c_j$
end

No.	Implementation	ROM (bytes)	Encryption $\overline{X}$ (μs)	Decryption $\overline{X}$ (μs)
1	Singh-1	1244	155.50	NA
2	Singh-2	992	208.25	NA
3	Szerwinski	868	185.92	NA
4	Optimized	2022	78.31	81.89
5	Protected	5526	$\approx 160 + (D \cdot 3.8)$	$\approx 160 + (D \cdot 4)$
6	Prot. (CMTF)	6324	$\approx 1183 + (D \cdot 56)$	NA

Table 7.1: Comparison of runtime and program size between the different TriCore AES implementations. The variable D denotes the number of dummy operations for the protected AES implementation.

Singh-1 implementation also uses tables lookups for the *xtimes* part of the *MixColumns* transformation. The runtime results of the measurements Singh-1 and Singh-2 fit with the results from Brijesh Singh presented in [Sin08].

The runtime of the protected AES implementation depends on the amount of dummy operations D and the number of AES runs which use the same mask for the S-box. The value in line five of Table 7.1 depicts the runtime, if the S-box is recalculated on every new AES run. With zero dummy operations one AES encryption takes 160 μs (100 kbyte/s).

Line six depicts the runtime for the CTMF approach. This implementation is just a concept. The decryption is not implemented. With this approach it is possible to remask every *SubBytes* operation. The runtime does not depend on the used masks since it does not recalculate the S-box. It calculates the inversion as we described it in Section 4.2.4. With $1183 + (D \cdot 56)\,\mu$s it is about eight times slower than the protected implementation in line five. This implementation can be used in higher order DPA resistant implementations.

Figure 7.1 and Figure 7.2 depict the runtime for $D = 0$ to $D = 255$. The S-box is recalculated at every new AES run. We can see a linear runtime behavior which is described by $160\,\mu\text{s} + D \cdot 3.8\,\mu\text{s}$ for the encryption. For the decryption, we get $160\,\mu\text{s} + D \cdot 4\,\mu\text{s}$, line five in Table 7.1. We can see, that the decryption takes a bit more time, which is mainly caused by the inverse *MixColumns* transformation. For the two figures, we measured the runtime for six different numbers of dummy operations, $D \in \{1, 50, 100, 150, 200, 255\}$ according to Algorithm 8 and drawed the regression line between the measured results. To calculate the polynomial for the regression line we use the Matlab function `polyfit` on the measured

samples.

To check, if the S-box recalculation works correctly, we build a version with $D = 10$ dummy operations and which recalculates the S-box after every 5^{th} AES run. The measured runtime of the first 50 runs is plotted in Figure 7.3. The peak denotes, that at this AES run the S-box has been recalculated. We can observe, that a recalculation of the S-box takes about $38\,\mu s$.

7.3 Timing Analysis

To measure the dependency between the runtime of an AES encryption and the input data five measurements sets were created. Between the different measurement sets, the key k and the plaintext p vary. Table 7.2 gives an overview about the key and plaintext pairs used.

Set	Plaintext	Key
1	$\{$00000000000000000000000000000000$\}_{16}$	$\{$000102030506070 80A0B0C0D0F101112$\}_{16}$
2	$\{$F658D966DA31D4A719096AB0209DBE4F$\}_{16}$	$\{$14151617191A1B1C1E1F202123242526$\}_{16}$
3	$\{$8A1E63C3C5F274CBBD8E26F0D91C179D$\}_{16}$	$\{$14151617191A1B1C1E1F202123242526$\}_{16}$
4	$\{$000102030405060708090A0B0C0D0E0F$\}_{16}$	$\{$3C3D3E3F41424344464748494B4C4D4E$\}_{16}$
5	$\{$000102030405060708090A0B0C0D0E0F$\}_{16}$	$\{$505152535556575 85A5B5C5D5F606162$\}_{16}$

Table 7.2: The five different measurement sets

All measurement sets contain random data, but set two and set three share the same keys and different plaintexts and set four and five use the same plaintext but different keys.

The runtime X_j of every encryption run (`aes_Encrypt`) was measured $1,000$ times for each set. Algorithm 9 denotes the measurement process.

Algorithm 9: Pseudo code for the timing analysis. The execution time of aes_Encrypt is measured and stored in X_j for every measurement set $i = 1, \ldots, 5$

```
   Input: k_1, ..., k_5, p_1, ..., p_5
1  for i = 1 to 5 do
2      for j = 1 to 1000 do
3          tic;
4          aes_Encrypt(k_i, p_i);
5          X_j = toc;
6      end
7  end
```

For the timing analysis, we need a higher resolution. Because we cannot increase the precision of the MCDS timing measurement, we decrease the CPU frequency instead. The CPU frequency is controlled by the PLL[1]. We have to setup the four main parameters: the P-Divider, N-Divider, K-Divider, and VCO[2] range where:

1 The Phase-Locked Loop (PLL) generate the CPU and system clock inside the TC1796.
2 Voltage Controlled Oscillator

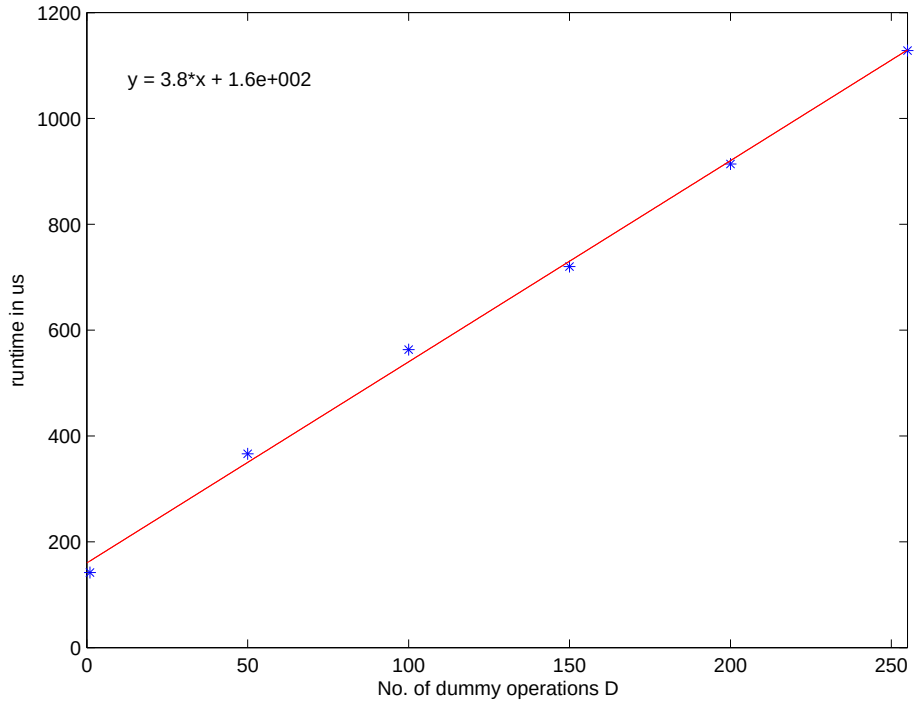

Figure 7.1: Runtime of the protected AES encryption with $1 \leq D \leq 255$. The S-box is recalculated after every AES encryption.

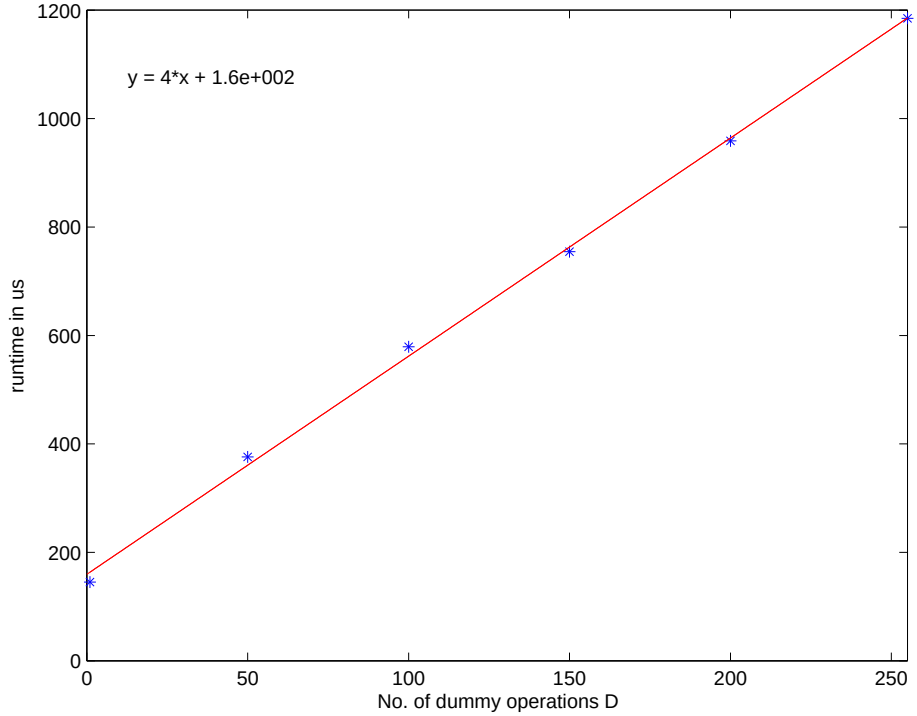

Figure 7.2: Runtime of the protected AES decryption with $1 \leq D \leq 255$. The S-box is recalculated after every AES decryption.

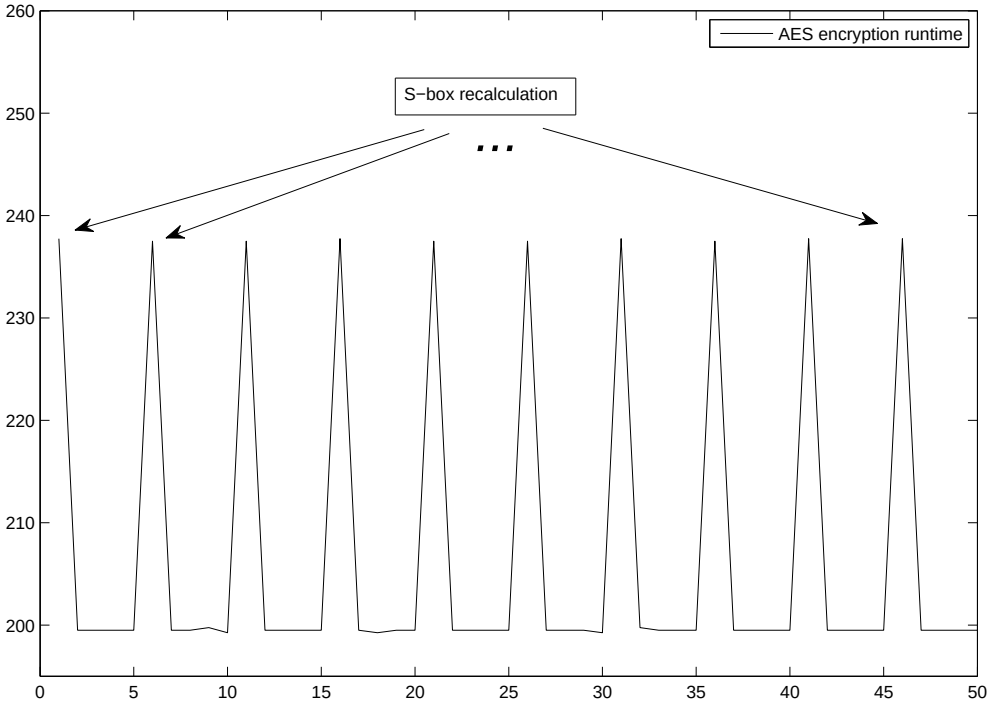

Figure 7.3: Runtime of the protected AES encryption with ten dummy operations (D=10). The S-box is recalculated after every 5^{th} AES encryption.

$$f_{CPU} = \frac{N}{P \cdot K} \cdot f_{OSC}. \tag{7.1}$$

With $N = 100$, $P = 5$ and $K = 16$ we get the smallest frequency $f_{CPU} = 25\,\text{MHz}$. The VCO range is set to 400 – 500 MHz.

The smallest measurable time unit Δ (the absolute failure) we can measure with the MCDS is $250\,\mu s$.To get a more precise time domain we oversample our measurements by the factor of 16^3, which means we average the measurement over 16 AES runs.

7.3.1 Timing Analysis of the Unprotected AES Implementations

Table 7.3 to Table 7.6 contain the arithmetic mean values $\overline{X}$, i. e., an estimator for the expected value μ_X, and the empirical standard deviation s_X, i. e., an estimator for σ_X, each for all $1,000$ measurements.

In Table 7.3 the timing behavior of the implementation from Szerwinski and in Table 7.4 our optimized AES implementation are depicted. Both leak timing information, since they show a different runtime for different combinations of the key and plaintext.

Set	Mean $\overline{X}$ in μs	Standard deviation s_X in μs
1	1119.46	0.0195204
2	1117.17	0.0186754
3	1117.26	0.0207298
4	1134.94	0.0216387
5	1130.26	0.0212278

Table 7.3: Runtime analysis of the AES implementation from Szerwinski

Set	Mean $\overline{X}$ in μs	Standard deviation s_X in μs
1	461.204	0.018
2	461.164	0.014
3	461.204	0.018
4	461.205	0.018
5	461.205	0.018

Table 7.4: Runtime analysis of the optimized AES encryption

The histogram in Figure 7.4 confirms this. We can distinguish the different runtimes for the different measurement sets. For example, we can clearly distinguish between set one on the left and set five on the right side of the histogram.

In Table 7.6, the measurement results from Singh-2 are given. In this AES implementation, the *MixColumns* transformation is calculated on-the-fly, as described in Section 2.3.3. It is the only transformation, with a conditional part (*xtimes*) during the calculation. As expected,

3 It would be enough to use oversampling by 10 to fulfill the sampling theorem, a further factor 6 is for higher precision.

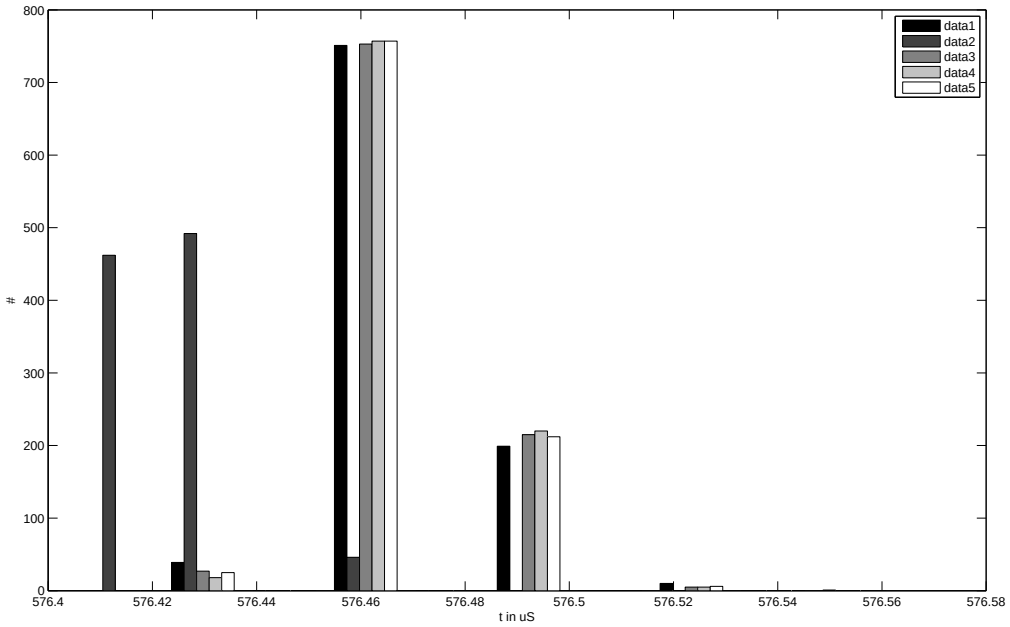

Figure 7.4: Histogram of the runtime for optimized AES implementation

this leads to an exploitable timing consumption. The mean values $\overline{X}$ depend on the processed data.

Set	Mean $\overline{X}$ in μs	Standard deviation s_X in μs
1	967.462	0.021
2	966.462	0.020
3	968.342	0.021
4	966.701	0.017
5	966.741	0.019

Table 7.5: Runtime analysis of the AES implementation from Singh-1

In Table 7.5, the measurement results from Singh-1 are shown. Also in this implementation, the mean values $\overline{X}$ for the five different measurements sets are distinguishable. The implementation from Singh-1 contains no obviously data dependent parts which would affect the runtime. Here the *xtimes* part of the *MixColumns* transformation is realized as a lookup table. We guess, that the different runtime results from platform depending scratchpad-RAM effects. This contradicts our assumption from Section 1.2, that a table lookup has a constant runtime. Thus we can observe a kind of cache effect.

Because the Protected AES implementation is based on the optimized AES implementation, it can be assumed, that the protected implementation also shows a different timing consumption on different key and plaintext pairs. It must be verified, that this information cannot be used during a timing analysis attack.

Set	Mean $\overline{X}$ in μs	Standard deviation s_X in μs
1	1282.25	0.020
2	1281.97	0.019
3	1281.29	0.020
4	1281.21	0.017
5	1279.69	0.020

Table 7.6: Runtime analysis of the AES implementation from Singh-2

7.3.2 Timing Analysis of the Protected AES Implementation

For the measurements we use $D = 16$ dummy operations. The *SubBytes* transformation is implemented as we described it in Section 6.6.1. We re-mask the S-box on every new run of the cipher.

Encryption

Table 7.7 depicts the mean values $\overline{X}$ for the different measurement sets, which also differs but with a much smaller variance.

Set	Mean $\overline{X}$ in μs	Standard deviation s_X in μs
1	1232.82	0.538
2	1232.80	0.540
3	1232.76	0.551
4	1232.80	0.560
5	1232.75	0.540

Table 7.7: Runtime of protected AES encryption with 16 dummy operations (D=16)

Figure 7.5 depicts the runtime of the an AES encryption with $D = 16$ dummy operations. The runtime was measured 1,000 times. As we expect, the most measured runtimes lie in the area $\overline{x} \pm s_x$.

The histogram in Figure 7.6 depicts that it is harder to distinguish between the measurement sets on the protected AES implementation.

To test, if the different measurement sets really do not lead to a distinguishable runtime, the two sample t-test is used. We fix the indices i and j. The statistical question we are interested in is if we can clearly distinguish between the sets X_k^i and Y_k^j, $k = 1, \ldots, 1000$. The t-test is applied among each mean value $X = X^{(i)}$, $i = 1, \ldots, 5$ and $Y = X^{(j)}$, $j = 1, \ldots, 5$, where:

- X_k – Runtime of `aes_Encrypt` in μs for set i, with k^{th} measurement, $k = 1, \ldots, 1000$
- Y_k – Runtime of `aes_Encrypt` in μs for set j, with k^{th} measurement, $k = 1, \ldots, 1000$
- Condition: X_k, Y_k i.i.d., $X_k \sim N(\mu_X, \sigma_X^2)$, $Y_k \sim N(\mu_Y, \sigma_Y^2)$, $k = 1, \ldots, 1000$, $\sigma_X^2 = \sigma_Y^2$ and unknown

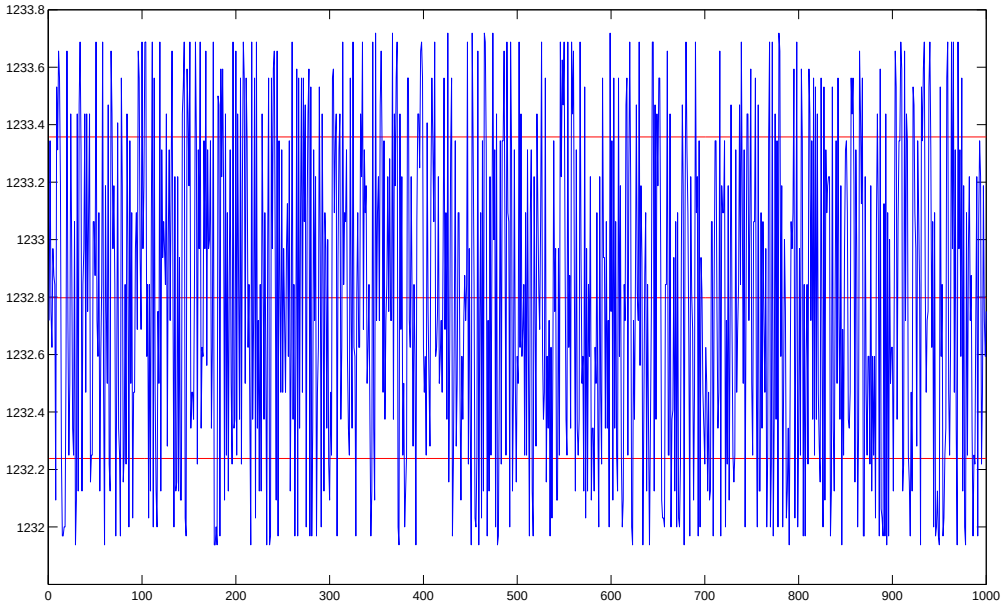

Figure 7.5: Runtime of the protected AES encryption (set four) with 16 dummy operations (D=16). The lines denote the mean (middle) and the variance of the runtime.

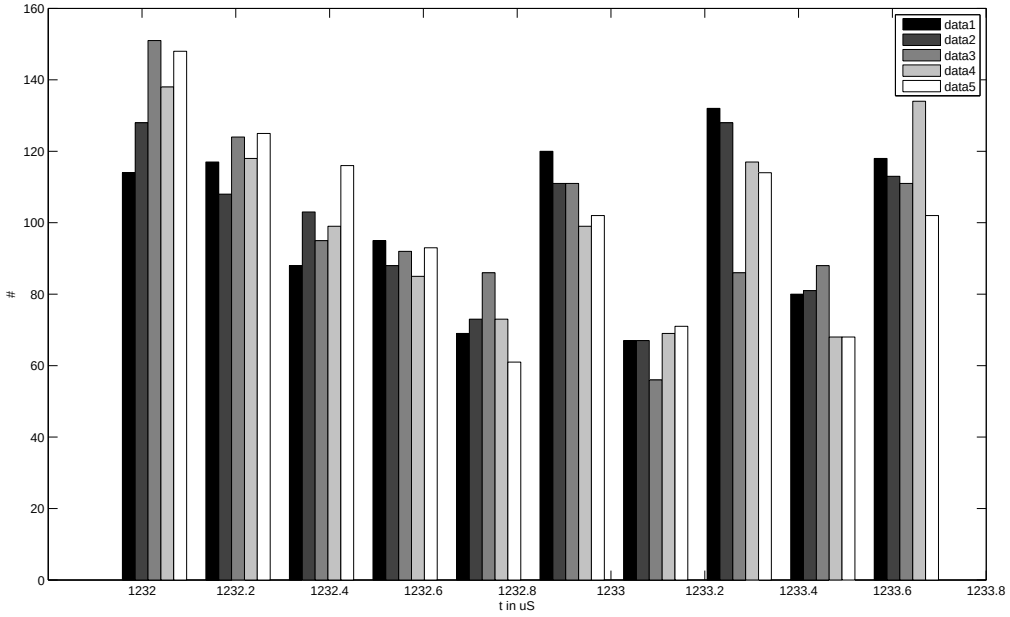

Figure 7.6: Histogram of the runtime for protected AES encryption with 16 dummy operations (D=16)

- Null hypothesis: $H_0 : \mu_X = \mu_Y$ versus $H_1 : \mu_X \neq \mu_Y$

Table 7.8 denotes the result for the protected AES implementation. The zero indicates that the test cannot be rejected. This means that the mean values come from the same population and are not distinguishable. This holds for every measurement set combination. We get more information from the p-value on the right hand side of Table 7.8.

> The p-value is the probability of observing the given result, [or one more extreme values] under the assumption that the null hypothesis is true. If the p-value is less than α, then you reject the null hypothesis. For example, if $\alpha = 0.05$ and the p-value is 0.03, then you reject the null hypothesis. The converse is not true. If the p-value is greater than α, you do not accept the null hypothesis. You just have insufficient evidence to reject the null hypothesis.
>
> (MATLAB Statistics Toolbox User's Guide)

j/i	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

j/i	1	2	3	4	5
1	0.099	0.537	0.702	0.210	
0.099	1	0.260	0.111	0.733	
0.537	0.260	1	0.715	0.465	
0.702	0.111	0.715	1	0.260	
0.210	0.733	0.465	0.260	1	

Table 7.8: Two sample t-test for the protected AES encryption with 16 dummy operations (left) and p-values of the test (right)

One pre-condition for the applicability of the two sample t-test is that the unknown variances of both random variables X and Y are the same. We can check this with Fisher's F-test. Fisher's F-test verifies if the variances σ_X^2 and σ_Y^2 of two random variables X and Y which are normally distributed with unknown and not necessarily equal mean values μ_x and μ_y are the same, see Section 3.1.3. Note, however, that the two-sample t-test checks statistical moments of first order, whereby the F-test checks statistical moments of second order. Effects of order two are more difficult to detect, in general, because they are smaller than effects of first order.

- X_k – Runtime of `aes_Encrypt` in μs for set i, with k^{th} measurement, $k = 1, \ldots, 1000$
- Y_k – Runtime of `aes_Encrypt` in μs for set j, with k^{th} measurement, $k = 1, \ldots, 1000$
- Condition: X_k, Y_k i.i.d., $X_k \sim N(\mu_X, \sigma_X^2)$, $Y_k \sim N(\mu_Y, \sigma_Y^2)$, $k = 1, \ldots, 1000$
- Null hypothesis: $H_0 : \sigma_x^2 = \sigma_y^2$ versus $H_1 : \sigma_x^2 \neq \sigma_y^2$

Table 7.9 denotes the F-test results for the measured timings. Every measurement set is compared with each other.

j/i	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

j/i	1	2	3	4	5
1	0.769	0.896	0.421	0.611	
0.769	1	0.870	0.279	0.828	
0.896	0.870	1	0.353	0.704	
0.421	0.279	0.353	1	0.200	
0.611	0.828	0.704	0.200	1	

Table 7.9: F-test for the protected AES encryption with 16 dummy operations (left) and the p-values of the test (right)

Set	Mean $\overline{X}$ in μs	Standard deviation s_X in μs
1	1268.25	0.545
2	1268.23	0.546
3	1268.19	0.560
4	1268.23	0.566
5	1268.18	0.547

Table 7.10: Runtime of protected AES decryption with 16 dummy operations (D=16)

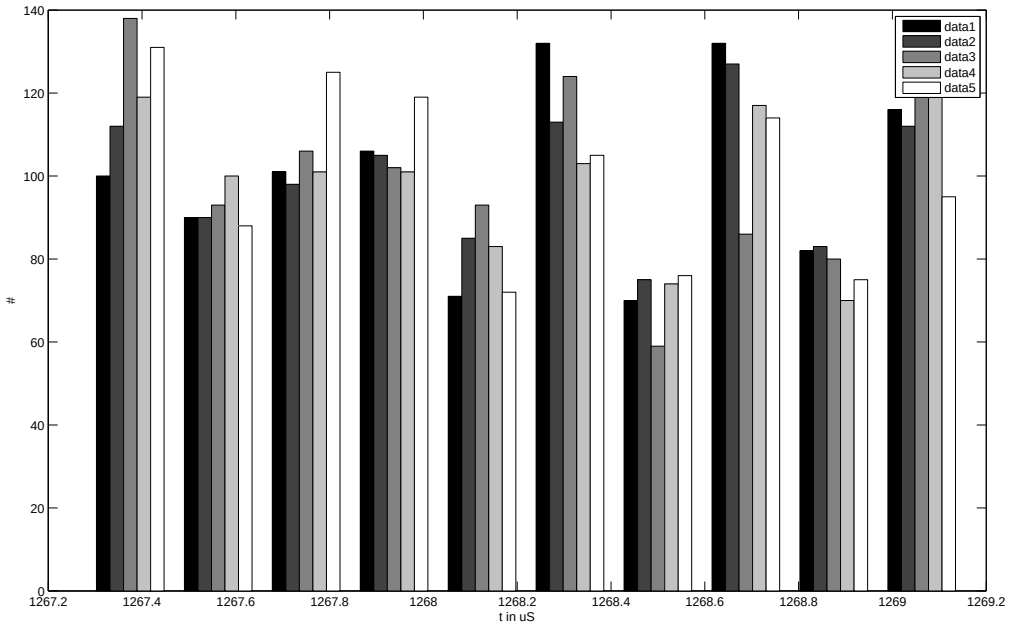

Figure 7.7: Histogram of the runtime for protected AES decryption with 16 dummy operations (D=16)

Decryption

Table 7.10 depicts the mean value $\overline{X}$ of the runtime for the given set. The mean values also differs for different key and plaintext combinations but with a much bigger variance.

The histogram in Figure 7.7 depicts that it is harder to make a differentiation between the measurement sets on the protected AES implementation. The appearance of a specific runtime for a specific measurement set is almost normal distributed over all possible measured timings.

To test, if the different measurement sets really do not lead to a distinguishable runtime, we also use the two sample t-test. The t-test is applied among each mean value $X = X^{(i)}$, $i = 1, \ldots, 5$ and $Y = X^{(j)}$, $j = 1, \ldots, 5$, where:

- X_k – Runtime of `aes_Decrypt` in μs for set i, with k^{th} measurement, $k = 1, \ldots, 1000$
- Y_k – Runtime of `aes_Decrypt` in μs for set j, with k^{th} measurement, $k = 1, \ldots, 1000$
- Condition: X_k, Y_k i.i.d., $X_k \sim N(\mu_X, \sigma_X^2)$, $Y_k \sim N(\mu_Y, \sigma_Y^2)$, $k = 1, \ldots, 1000$, $\sigma_X^2 = \sigma_Y^2$ and unknown
- Null hypothesis: $H_0 : \mu_X = \mu_Y$ versus $H_1 : \mu_X \neq \mu_Y$

Table 7.11 denotes the result for the decryption. The zero indicates that the hypothesis cannot be rejected. This means that the mean values probably come from the same population and are not distinguishable. This holds for every measurement set combination.

j/i	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

j/i	1	2	3	4	5
1	0.088	0.534	0.674	0.220	
0.088	1	0.241	0.096	0.698	
0.534	0.241	1	0.726	0.481	
0.674	0.096	0.726	1	0.276	
0.220	0.698	0.481	0.276	1	

Table 7.11: Two-sample t-test for the protected AES decryption with 16 dummy operations (left) and the p-values of the test (right)

The condition for the t-test is, that the unknown variance of both random variables X and Y is the same. To ensure this, we use the F-test again.

- X_k – Runtime of `aes_Decrypt` in μs for set i, with k^{th} measurement, $k = 1, \ldots, 1000$
- Y_k – Runtime of `aes_Decrypt` in μs for set j, with k^{th} measurement, $k = 1, \ldots, 1000$
- Condition: X_k, Y_k i.i.d., $X_k \sim N(\mu_X, \sigma_X^2)$, $Y_k \sim N(\mu_Y, \sigma_Y^2)$, $k = 1, \ldots, 1000$
- Null hypothesis: $H_0 : \sigma_x^2 = \sigma_y^2$ versus $H_1 : \sigma_x^2 \neq \sigma_y^2$

Table 7.12 denotes the F-test results for the measured timings. We can see, that they are all above the 5%.

The statistical tests show, that we cannot distinguish between the different sets, i. e., the different keys, on hand of the runtime behavior of our protected AES implementation. Thus, we can assume that the protected AES implementation is resistant against timing analysis.

j/i	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

j/i	1	2	3	4	5
1	0.786	0.877	0.344	0.564	
0.786	1	0.907	0.230	0.757	
0.877	0.907	1	0.275	0.672	
0.344	0.230	0.275	1	0.140	
0.564	0.757	0.672	0.140	1	

Table 7.12: Fishers F-test for the protected AES decryption with 16 dummy operations (left) and p-values of the test (right)

7.4 Power Analysis

It is not in the scope of this thesis to analyze the protected AES implementation with respect to Power Analysis resistance. The design of the countermeasures was chosen in such a way that the implementation *should* be resistant against Simple Power Analysis and standard Correlation and Differential Power Analysis, see [HOM06]. This means that more traces are needed for a successfully mounted DPA attack than for an attack against an unprotected implementation. However, this implementation is susceptible to advanced power analysis techniques, especially second order attacks.

The theoretical results from [THM07] and the practical results from [TH08] showed, that the number of required traces for a successful mounted SCA increases by $(D + 1) \cdot 16$. The number of traces which are needed to attack an unprotected implementation on the TriCore has not been investigated yet.

To measure the resistance against a DPA, one should start to build a measurement setup. The setup could be tested by analyzing an unprotected AES version like the optimized AES implementation. This implementation should be fairly easy to attack.

Afterwards the protected AES implementation should be attacked with methods like windowing and second order DPA, see Section 4.3.1. However, this requires a fair amount of experience in attacking protected implementations.

Chapter 8

Conclusion

The topic of this master thesis was the side-channel resistant implementation of the Advanced Encryption Standard (AES) on a typical automotive processor.

Symmetric crypto primitives are the basis for most cryptographic architectures. The Advanced Encryption Standard (AES) is *the current standard* for block ciphers. An efficient *and* secure implementation of this algorithm therefore is very important for future security mechanisms in the automotive world. While there are many fairly efficient AES implementations in the field, most of them are not optimized for embedded systems, especially in the automotive world. Most AES implementations are for 8-bit systems, e.g., smart cards, or for 32-bit architectures with lots of memory. However, in the automotive domain we often have 32-bit RISC processors and tight constraints on the memory and the program size while speed is most often not an issue. Therefore, the selection and implementation of an unprotected AES specially adapted for the needs of the automotive industry is necessary.

The topic of side-channel attacks is one of the most important issues in applied cryptography. In the literature, one can find many results on high security systems like smart cards. However, until now we are not aware of any side-channel analysis resistant implementation of AES on automotive processors. For the first time, an efficient (32-bit optimized) *and* secured (countermeasures against timing and power analysis) AES implementation has been investigated on automotive platforms.

The target platform for the implementation was an Infineon TriCore TC1796. This processor is widely used in Engine Control Units (ECU) made by Robert Bosch GmbH as well as by Continental AG, for example. It is a 32-bit RISC processor.

After a short introduction of the Advanced Encryption Standard we discussed in Chapter 2 various implementation options for unsecured implementations. Since many of the existing highly efficient implementations are very processor-specific and are therefore not directly comparable, we introduced a simple processor model for comparison. Note that the resulting estimates are not very meaningful for itself but are important with respect to the other implementations. Thus, not the absolute number of clock cycles and memory consumption is important but the relative factor compared to other implementations. Specifically, we estimated a faster runtime with factor 2.2 between an existing, optimized 8-bit AES implementation(3808 cc) and a 32-bit optimized implementation (1732 cc).

In practice, we could achieve this factor with our real optimized AES implementation (78.31 μs) which utilizes the TriCore instruction set and uses the whole 32-bit register size. Compared to the optimized 8-bit implementation (Singh-2 with 208.3 μs) our implementation is faster by the factor 2.6 as the reference implementation.

Since the overall security level of automotive processors is moderate (no secure memory, for example) it should be enough to secure an AES implementation against the most dangerous,

low-level attacks, i.e., Timing Analysis (TA), Simple and Differential Power Analysis (SPA, DPA) (first order). In Chapter 3 we therefore discussed various attack techniques for Side Channel Analysis (SCA) and gave an overview about software countermeasures against timing analysis and (first order) Differential Power Analysis (DPA).

With our goal, an efficient and protected AES implementation, in mind we analyzed the existing SCA countermeasures in terms of performance, memory requirement and known attacks in Chapter 4. Using our processor model, we showed that AES implementations with resistance against second order DPA attacks require, in general, over ten times more clock cycles than unprotected implementation. Irrespective of the security aspects, overheads of this order are too much for an implementation on automotive processors. So we decided to use a first order DPA resistant masking method and combine it with randomization (shuffling and dummy states) as proposed in [HOM06] and [THM07]. In these two papers, the authors carefully selected various low-cost countermeasures to achieve the overall security level required.

In addition to this method, we provide a concept study for a masking scheme called CMTF (Combined Masking in Tower Fields), that can be used in an implementation that is resistant against higher order DPA attacks. Originally, CMTF has been proposed as hardware countermeasure. However, [OS05] used precomputed tables for the transformation between sub-fields, making it thus applicable as a software countermeasure. The CMTF method with precomputed tables can be combined with the randomization approach by Herbst et al., i. e., shuffling and dummy states).

It has turned out, that the affine transformation which is part of the *SubBytes* transformation is a bottleneck in terms of runtime because it works bitwise on each byte. An improvement would be if four affine transformation would be calculated on four bytes of the State matrix in parallel. This would require the possibility of rotating the four bytes in a word in parallel. The CMTF masking scheme as well as the other scheme which calculate the *SubBytes* transformation on the fly can be used in implementations which use more than one mask for the bytes of the State matrix, since the RAM consumption of the additive masking increases very fast with the number of masks used.

In Chapter 7 we evaluated the timing behavior of our implementations. We showed, that the unprotected implementations are vulnerable to timing analysis attacks. In addition we showed, that our protected implementation is probably timing analysis resistant. Power analysis has been left as a subject for further investigation.

In our theoretical part, we estimated a runtime slowdown by a factor of 1.9 between the 32-bit optimized implementation (1732 cc) and the additive masked implementation (3332 cc). We could observe this factor in practice at the real optimized implementation ($\approx 78\,\mu$s) and the protected implementation with no dummy operations ($\approx 160\,\mu$s). This gives us a factor of two. The security level can be further increased by introducing additional dummy states ($D \neq 0$).

Summarizing, for the first time we have an efficient and side-channel secured AES implementation on a typical automotive processor. The resulting implementation is very competitive in terms of program and memory size as well as performance. The security of the implementation is adapted to the current security level of automotive processors (low-cost countermeasures since we have no secure memory or tamper protection). The implementation can be used as a plug and play solution which can replace vulnerable unprotected AES implementations.

We predict, that in the mid-term future, no automotive manufacturer or supplier can afford

to implement SCA-vulnerable cryptographic algorithms. We have made a first step into this direction.

Bibliography

[AG01] M. L. Akkar and Chr. Giraud. An Implementation of DES and AES, Secure against Some Attacks. In C. K. Koç, D. Naccache, and Chr. Paar, editors, *Proceedings of CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer-Verlag, 2001.

[AK98] Ross J. Anderson and Markus G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 125–136, London, UK, 1998. Springer-Verlag.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *ASIACCS 2007: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320, New York, NY, USA, 2007. ACM.

[APSQ06] Cédric Archambeau, Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Template Attacks in Principal Subspaces. In Louis Goubin and Mitsuru Matsui, editors, *Proceedings of CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 2006.

[BBF+03] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient Software Implementation of AES on 32-bit Platforms. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Proceedings of CHES 2002: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 2523 of *Lecture Notes in Computer Science*, pages 159–171. Springer-Verlag, 2003.

[Ber05] Daniel J. Bernstein. Cache-timing Attacks on AES, 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[BGK04] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably Secure Masking of AES. In H. Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography – SAC 2004*, number 3357 in Lecture Notes in Computer Science, pages 69–83. Springer-Verlag, 2004.

[Bos98] Karl Bosch. *Statistik-Taschenbuch.* Oldenbourg, January 1998.

[BS08] Daniel Bernstein and Peter Schwabe. New AES Software Speed Records. Cryptology ePrint Archive, Report 2008/381, 2008. `http://eprint.iacr.org/`.

[BSI01] Bundesamt für Sicherheit in der Informationstechnik BSI. AIS 20: Funktionalitätsklassen und Evaluationsmethodologie für deterministische Zufallszahlengeneratoren. Anwendungshinweise und Interpretationen zum Schema (AIS) Version 1,

2.12.1999, BSI, 2001. Available at `http://www.bsi.bund.de/zertifiz/zert/interpr/ais20.pdf`.

[CB09] D. Canright and Lejla Batina. A Very Compact "Perfectly Masked" S-Box for AES (corrected). Cryptology ePrint Archive, Report 2009/011, 2009. `http://eprint.iacr.org/`.

[CCD00] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential Power Analysis in the Presence of Hardware Countermeasures. In Christof Paar and Çetin Kaya Koç, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer-Verlag, 2000.

[CGPR08] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, and Matthieu Rivain. Attack and Improvement of a Secure S-Box Calculation Based on the Fourier Transform. In *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 2008.

[Cha82] David Chaum. Blinding Signatures for Untraceable Payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology – CRYPTO 1982*, pages 199–203, 1982.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In *Proceedings of CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28, London, UK, August 2002. Springer-Verlag.

[Den07] Tom St Denis. *Cryptography for Developers*. Syngress Media, 2007.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael, AES – The Advanced Encryption Standard*. Springer-Verlag, 2002.

[FIP01] FIPS-197. *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197, November 2001. Available at `http://csrc.nist.gov/`.

[Gla01] Brian Gladman. A Specification for Rijndael, the AES Algorithm. v3.1.0, `http://fp.gladman.plus.com/cryptography_technology/rijndael/aes.spec.v310.pdf`, 2001.

[Gla07] Brian Gladman. A Specification for Rijndael, the AES Algorithm. v3.1.6, `http://fp.gladman.plus.com/cryptography_technology/rijndael/aes.spec.v316.pdf`, 2007.

[GLRP06] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. Stochastic Methods. In Louis Goubin and Mitsuru Matsui, editors, *Proceedings of CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 15–29. Springer-Verlag, 2006.

[GT03] Jovan Dj. Golic and Christophe Tymen. Multiplicative Masking and Power Analysis of AES. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Proceedings of CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 198–212. Springer-Verlag, 2003.

[HJM05] Martin Hell, Thomas Johansson, and Willi Meier. Grain - A Stream Cipher for Constrained Environments. eSTREAM, ECRYPT Stream Cipher. Technical report, 2005/010, ECRYPT (European Network of Excellence for Cryptology), 2005.

[HKQ99] Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. cAESar Results: Implementation of Four AES Candidates on Two Smart Cards. In Morris Dworkin, editor, *Proceedings of Second Advanced Encryption Standard Candidate Conference*, pages 95–108, Rome, Italy, March 22-23, 1999, 1999.

[HMS00] Erwin Hess, Bernd Meyer, and Torsten Schütze. Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures – A Survey. In *EUROSMART Security Conference*, pages 55–64, 2000.

[HOM06] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *Proceedings of ACNS 2006*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–255. Springer-Verlag, 2006.

[IAI06] IAIK-2006. VLSI Products – Software Modules. `http://www.iaik.tugraz.at/research/vlsi/02_products/index.php`, January 2006.

[Inf02] Infineon Technologies AG. *TriCore 2, 32-bit Unified Processor Core*, 2002,08 edition, Sept 2002.

[Inf05] Infineon Technologies AG. *TriCore™, Instruction Set Simulator (ISS), User Guide*, 2005-01 edition, January 2005.

[Inf07] Infineon Technologies AG. *TC1796 32-Bit Single-Chip Microcontroller, TriCore, User's Manual*, 2007-07 edition, July 2007.

[Inf08a] Infineon Technologies AG. *TC1796 32-Bit Single-Chip Microcontroller, TriCore, Data Sheet*, 2008-04 edition, April 2008.

[Inf08b] Infineon Technologies AG. *TC1796 32-Bit Single-Chip Microcontroller, TriCore, Instruction Set, V1.3 & V1.3.1 Architecture*, 2008-01 edition, July 2008.

[JPS05] Marc Joye, Pascal Paillier, and Berry Schoenmakers. On Second-Order Differential Power Analysis. In *CHES 2005: Revised Papers from the 7th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 3659 of *Lecture Notes in Computer Science*, pages 293–308. Springer-Verlag, 2005.

[Ker83] Auguste Kerckhoffs. La cryptographie militaire. In *Journal des sciences militaires*, volume IX, pages 5–83, Jan. 1883.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *Proceedings of CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[Kna06] Anthony W. Knapp. *Basic Algebra*. Birkhäuser, 2006.

[Koc96] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Proceedings of CRYPTO 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[Mat07] The MathWorks, Inc. *Statistics Toolbox User's Guide, Version 6.1*, 2007.

[MDS99] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Investigations of Power Analysis Attacks on Smartcards. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 151–162, Berkeley, CA, USA, 1999. USENIX Association.

[Mes00] Thomas Messerges. Securing the AES Finalists Against Power Analysis Attacks. In Bruce Schneier, editor, *Proceedings of FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 2000.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007. `http://www.dpabook.org`.

[MS03] Sumio Morioka and Akashi Satoh. An Optimized S-box Circuit Architecture for Low Power AES Design. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 2523 of *Lecture Notes in Computer Science*, pages 172–186. Springer-Verlag, 2003.

[NIS98] NIST. Known Answer Test for Advanced Encryption Standard (AES). `http://csrc.nist.gov/archive/aes/rijndael/rijndael-vals.zip`, 1998.

[NS06] Michael Neve and Jean-Pierre Seifert. Advances on Access-Driven Cache Attacks on AES. In *Selected Areas in Cryptography*, pages 147–162, 2006.

[OMP04] Elisabeth Oswald, Stefan Mangard, and Norbert Pramstaller. Secure and Efficient Masking of AES – a Mission Impossible? Cryptology ePrint Archive, Report 2004/134, 2004. `http://eprint.iacr.org/`.

[OMPR05] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A Side-Channel Analysis Resistant Description of the AES S-Box. In Henri Gilbert and Helena Handschuh, editors, *Proceedings of FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 413–423. Springer-Verlag, 2005.

[OS05] Elisabeth Oswald and Kai Schramm. An Efficient Masking Scheme for AES Software Implementations. In JooSeok Song, Taekyoung Kwon, and Moti Yung, editors, *Proceedings of WISA 2005*, number 3786 in Lecture Notes in Computer Science, pages 292–305. Springer-Verlag, 2005.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *Proceedings of CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2006.

[Paa94] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. Dissertation, Institute for Experimental Mathematics, Universität Essen, Deutschland, 1994. `http://crypto.rub.de/imperia/md/content/texte/theses/paar_php_diss.pdf`.

[RÖ3] Christian Röpke. Praktikum B: Embedded Smartcard Microcontrollers. `http://www.christianroepke.de/studium_praktikumB.html`, 2003.

[RDJ+01] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient Rijndael Encryption Implementation with Composite Field Arithmetic. In *CHES 2001: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, pages 171–184. Springer-Verlag, 2001.

[Rij01] Vincent Rijmen. Efficient Implementation of the Rijndael S-Box. Technical report, Katholieke Universiteit Leuven, Dept. ESAT, Belgium, 2001.

[Seu05] Hermann Seuschek. DPA-Analyse von Implementierungen symmetrischer kryptographischer Algorithmen. Diplomarbeit, Technische Universität München, 2005.

[Sin08] Brijesh Singh. Bosch Cryptographic Library – Documentation. Technical report, Robert Bosch GmbH, CR/AEA, 2008.

[SMTM01] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In Colin Boyd, editor, *Proceedings of ASIACRYPT '2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer-Verlag, 2001.

[Sti05] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC press, 3rd edition, 2005.

[TH08] Stefan Tillich and Christoph Herbst. Attacking State-of-the-Art Software Countermeasures—A Case Study for AES. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *LNCS*, pages 228–243. Springer, 2008.

[THM07] Stefan Tillich, Christoph Herbst, and Stefan Mangard. Protecting AES Software Implementations on 32-Bit Processors Against Power Analysis. In Jonathan Katz and Moti Yung, editors, *Proceedings of ACNS 2007*, volume 4521 of *Lecture Notes in Computer Science*, pages 141–157. Springer-Verlag, 2007.

[Tri03] Elena Trichina. Combinational Logic Design for AES Subbytes Transformation on Masked Data. Technical report, Cryptology ePrint Archive: Report 2003/236, 2003.

[TSG02] 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, *Proceedings of CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 187–197. Springer-Verlag, 2002.

[WG04] Guido Walz and Barbara Grabowski, editors. *Lexikon der Statistik: mit ausführlichem Anwendungsteil (German Edition)*. Spektrum Akademischer Verlag, first edition, June 2004.

[WOL02] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC Implementation of the AES S-Boxes. In *Proceedings of CT-RSA 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 67–78. Springer-Verlag, 2002.