Tamper Resistance of AES
—
Models, Attacks and Countermeasures

A dissertation submitted to the
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF PADERBORN

for the degree of
Doktor der Naturwissenschaften

presented by
VOLKER KRUMMEL

accepted on the recommendation of
Prof. Dr. Johannes Blömer, examiner
Prof. Dr. Joachim von zur Gathen, co-examiner

2007

≪Timmy & Finn – Sonnenkinder, die auch im Regen lachen≫

Acknowledgments

I am deeply grateful to my supervisor, Prof. Dr. Johannes Blömer, for his great support and continuous encouragement in writing this thesis. Among other topics, he introduced me into the field of tamper resistance and side channel attacks and supplied me with new interesting and challenging problems and ideas. Johannes allowed me great freedom to do my research and he always took time to discuss the ongoing progress. His comments and suggestions were always very helpful to improve my work.

I am also truly indebted to my second supervisor, Prof. Dr. Joachim von zur Gathen, who sparked my interest in cryptography. The opportunity to join his working group allowed me to deepen my research in this fascinating area.

Furthermore, I would like to thank Dr. Jean-Pierre Seifert, the coordinator of our joint project with the Intel Corporation. The cooperation with Intel not only implied financial support of my research but also provided valuable insights in recent cryptographic problems.

This thesis would not have been possible without the generous support of the “Institut für Industriemathematik” of the University of Paderborn. Special thanks go to Tanja Bürger and Dr. Robert Preis who were very helpful in handling all the administrative obstacles.

For proof reading parts of my thesis, I would like to thank Marcel R. Ackermann, Dr. Valentina Damerow and Stefanie Naewe.

Contents

1 Introduction **1**

2 The Advanced Encryption Standard (AES) **5**

2.1 Symmetric Block Ciphers 5

2.2 Basic Algebraic Structures of AES 6

2.2.1 Representation of Data 6

2.2.2 The Finite Field $\mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x + 1\rangle$ 7

2.2.3 The Ring $\mathbb{F}_2[x]/\langle x^8 + 1\rangle$ 8

2.2.4 The Ring $\mathcal{R} = \mathbb{F}_{256}[y]/\langle y^4 + 1\rangle$ 9

2.3 The Standard Implementation of AES 9

2.3.1 State Transformations 10

2.3.2 Encryption 12

2.3.3 Key Expansion 12

2.3.4 Decryption 14

2.4 The Fast Implementation of AES 16

3 Security and Side Channel Attacks **19**

3.1 General Principles of Side Channel Attacks 20

3.2 Side Channels 21

3.2.1 Timing Attack 21

3.2.2 Power Analysis 23

3.2.3 Fault Attacks 23

3.2.4 Cache Attacks 23

3.2.5 Other Side Channel Attacks 24

3.3 Countermeasures 24

4 Provably Secure Randomization of Cryptographic Algorithms 25

4.1 Security Model 28

4.1.1 Discussion of the Security Notion 30

4.2 Masking AES 31

4.3 Perfectly Masking AES against Order-1 Adversaries 33

4.3.1 Idea 33

4.3.2 Method 34

4.3.3 Security Analysis 35

4.3.4 Simplified Version 37

4.4 Implementation and Costs 38

4.4.1 Efficient Hardware Implementation over $GF(((2^2)^2)^2)$ 38

4.4.2 Cost and Comparison to Previous Countermeasures 39

4.5 Order-d Perfectly Masking 41

4.5.1 Perfect Mask Change 41

4.5.2 Squaring 46

4.5.3 Multiplication 46

4.6 Conclusions 47

5 Fault Based Collision Attacks 49

5.1 The Concept of Fault Attacks 52

5.1.1 Methods to Induce Faults 52

5.1.2 Fault Models 53

5.2 The Concept of Collision Attacks 56

5.3 New Fault Model 56

5.3.1 Notation 56

5.3.2 Model 57

5.4 Fault Based Collision Attacks on AES 59

5.4.1 Basic Attack 60

5.4.2 Second Attack 61

5.4.3 Third Attack 64

5.4.4 Fourth Attack 67
5.4.5 Fifth Attack 67
5.5 Conclusion 69

6 Cache Behavior Attacks (CBAs) 71
6.1 Cache Mechanism and Technical Background 73
6.2 Security Models for CBAs 75
6.2.1 Fundamental Model for CBAs 76
6.2.2 Time Driven CBA 77
6.2.3 Trace Driven CBA 80
6.2.4 Access Driven CBA 82
6.2.5 Extending the Threat Model for Access Driven CBAs 84
6.3 Access Driven CBAs on AES 85
6.3.1 Access Driven CBA on the First Round 85
6.3.2 Access Driven CBA on the Last Round 87
6.4 General Methods to Thwart CBAs 88
6.5 Information Leakage and Resistance 89
6.6 Information Leakage and Resistance of Selected Implementations 92
6.7 Countermeasures Based on Permutations 100
6.7.1 An Access Driven CBA on a Permuted Sbox 101
6.7.2 Separability and Distinguished Permutations 103
6.8 Summary of Countermeasures and Open Problems 106

A Sbox Tables $\mathbf{T}_0, \ldots, \mathbf{T}_4$ of AES 109

B Decompositions of the AES Sbox 115

Chapter 1

Introduction

Security in whatsoever context or meaning is the goal of human beings ever since the dawn of mankind. One aspect of security is the secret communication, i.e., preventing others from reading private messages. The oldest approach in making texts hard to read dates back to about 4000 years. At that time in Egypt, a master scribe used unusual hieroglyphs to obfuscate the meaning of an inscription in the tomb of Khnumhotep II (Kahn 1996). Cryptography – the science of secret writing – was born.

In the course of time, people invented a lot of systems for keeping messages secret. Most of them were broken because of the lack of thorough analysis and invalid assumptions of the inventor. A famous example was the Enigma cipher machine used by German forces in the Second World War. Several drawbacks of the Enigma in connection with a bad protocol and protocol failures of the participants helped Polish and British experts to break the cipher. Since World War 2, people understand the importance of cryptography and the theory of the design and the analysis of encryption algorithms made enormous progress. Nowadays we have a large number of strong algorithms whose security was analyzed independently by crypto researchers all over the world, e.g., see (Menezes, van Oorschot and Vanstone 1997) and (Schneier 1996).

But cryptography expanded from the science of secret writing to the science of arbitrary security problems like authentication or data integrity. Cryptography can solve some very difficult problems concerning security. Hence, cryptographic algorithms are the main building blocks of security systems like access control or electronic payments. However, it was known right from the beginning that using strong cryptographic algorithms does not necessarily lead to a secure system. Quite the contrary is true. Using weak cryptography would not weaken many systems because there are several other components of the systems that allow even easier attacks. We are confronted with this kind of problem when we see the security problems that occur because of the human factor or implementational mistakes like buffer overflows etc. Securing a system can be compared to protecting a house against burglars. Further strengthening the front door with sophisticated locks does not improve the security if the window on the back is still open. An attacker is not fair. He would not spend his (life)time trying to pick the locks of the front door but simply slips in through the open window. The same is true for security systems and even for cryptographic algorithms. We cannot expect that an attacker does what we suppose him to do. He will take every chance he can get to break the system. Since the system is only as secure as its weakest link, to improve the security of a system one has to perform the following steps according to (Ferguson and Schneier 2003):

1. detect all links
2. determine weak links
3. strengthen weak links

These steps are easily written down but very hard to perform. I.e., detecting all links and determining weak links is very hard and tricky. The problem is that there do not exist any rules an attacker sticks to.

Peter Wright was the first who published details about operation "ENGULF" an example for such an "unfair" attack (Wright 1987). Wright was a scientist at the MI5, one of the secret services of the United Kingdom. During the Suez crisis in 1956, the MI5 was interested in the messages of the Egyptian embassy that were encrypted by a Hagelin rotor machine. To improve the secrecy a new key was set up every day. Although the MI5 had exactly the same model of the cipher machine they could not break the encryption efficiently. Therefore, Wright suggested to place a microphone close to the cipher machine to determine the key settings by listending to the sound that occurs when setting up a new key. The sound enabled the MI5 to figure out the daily key and read all the messages.

In 1985 van Eck published a different approach later called "van Eck phreaking" to obtain private information (van Eck 1985). He showed how to exploit the electromagnetic emanations of computer displays to reconstruct the content of the display even from a large distance.

Attacks that bypass security mechanisms by exploiting additional information or by manipulating the environment are called *tampering attacks*. These attacks show that security engineering – the science of developing reliable and secure systems – is a much wider field than cryptography, e.g., see (Anderson 2001).

But cryptography itself became a target of security concerns when Kocher published an attack that determines a secret RSA key by analyzing the running times of encryptions (Kocher 1996). Only a few years later, he also showed how to break cryptographic schemes by analyzing the power consumption (Kocher, Jaffe and Jun 1999). Several similar methods – so called *side channel attacks* – were developed to break cryptographic algorithms very efficiently. As strengthening the links of a security system, protecting cryptographic algorithms against side channel attacks is quite tricky.

Organization of the Thesis and Main Results In this thesis we focus on analyzing the tamper resistance of cryptographic algorithms. More precisely, we examine the security of todays most important symmetric encryption scheme, the Advanced Encryption Standard (AES), against side channel attacks.

The first goal was to develop a general and strong model in which the effectiveness of countermeasures to thwart side channel attacks can be analyzed. This goal was motivated by finding a secure implementation of AES, a problem that was not satisfyingly solved before. In Chapter 4 we present our strong and general model which covers adversaries of different power. After that, we develop a general method to implement ciphers like AES provably secure in our model. We give the security proof together with a thorough analysis of the costs of our AES implementation in hardware. The results of this chapter were published in (Blömer, Guajardo and Krummel 2004).

A further goal was to analyze the effectiveness of countermeasures that were proposed to thwart side channel attacks but were not analyzed thoroughly. We focus on the so called memory encryption, a method that is based on encrypting the main memory to prevent information leakage. At first sight, memory encryption provides a large improvement of security and hence is used in many high security smartcards. In Chapter 5 we show that this first impression is wrong. We present a new concept of fault attacks called *fault based collision attacks* that defeats memory encryption using only a moderate number of faults. The results of this chapter were published in (Blömer and Krummel 2006).

In the last part of the thesis we analyze a different kind of side channel attacks, so called *cache based attacks*. Cache based attacks have been proven to be very powerful and turned out to be one of the biggest threats of cryptographic software implementations running on computers with cache. In Chapter 6 we first strengthen the existing threat model to adapt it to the recent methodology of cache based attacks. We introduce two security concepts *information leakage* and *resistance*. Information leakage measures the maximal amount of information that leaks through an arbitrary number of cache based attacks. The resistance estimates the information an attacker may get after a single cache based attack. We analyzed several implementations of AES determining their information leakage and their resistance. It turns out that all implementations proposed so far provide only poor resistance and leak all key bits. Therefore, we propose a new implementation of AES using small sboxes that does not leak a single key bit. Furthermore, we analyzed a proposed countermeasure based on random permutations. We show how to efficiently defeat this countermeasure using cached based attacks. To improve the effectiveness of this countermeasure we develop a special class of permutations so called *distinguished permutations*. Using distinguished permutations we can provably protect half of the key bits even for an unlimited number of cache attacks. The results of this chapter were published in (Blömer and Krummel 2007).

Chapter 2

The Advanced Encryption Standard (AES)

In 1977, the National Bureau of Standards (NBS) of the USA announced the first standardized symmetric encryption algorithm called *Data Encryption Standard* (DES) which immediately became the de facto standard worldwide. In 1997, the National Institute of Standards and Technology (NIST), formerly named NBS, started to search a successor of DES. The NIST arranged a public competition of proposed algorithms that were submitted by several researchers of the cryptography community. These submissions where publicly analyzed by crypto researchers all over the world. Five candidate algorithms made it to the final decision. In the end Rijndael, an algorithm of the two Belgian cryptographers Joan Daemen and Vincent Rijmen, was chosen to be the successor of DES named the Advanced Encryption Standard (AES). In this chapter we first give the background of symmetric encryption algorithms and then describe the AES in more detail. Further information about the AES can be found in (Daemen and Rijmen 2002) and (NIST 2001). A more condensed description of the AES can be found in (Lenstra 2002).

2.1 Symmetric Block Ciphers

Since the seminal paper (Diffie and Hellman 1976) encryption schemes (or ciphers) can be classified as either symmetric or asymmetric ciphers. Asymmetric ciphers use a pair of keys, a public key for encryption and a private key for decryption. For the security of this kind of encryption systems it is essential that the private key cannot be derived from the public key efficiently. Using a key pair (a public and a private one) allows two parties to communicate privately without sharing a common secret. Two famous examples for asymmetric ciphers are RSA (Rivest, Shamir and Adleman 1978) and the ElGamal cryptosystem (ElGamal 1985).

Symmetric ciphers only deal with a single key for both, encryption and decryption. Hence, before being able to communicate securely both parties have to agree on a common secret key. To be more precise, a symmetric encryption scheme is defined as follows.

Definition 1 (symmetric encryption scheme) *Let $\mathcal{P}$, $\mathcal{K}$ and $\mathcal{C}$ be the sets of valid plaintexts, keys and ciphertexts respectively. A* symmetric encryption scheme *consists of a pair of algorithms (enc, dec). The algorithm enc computes the unique ciphertext $c \in \mathcal{C}$ given a valid plaintext $p \in \mathcal{P}$ and a valid key $k \in \mathcal{K}$:*

$$\begin{aligned} enc : \mathcal{P} \times \mathcal{K} &\rightarrow \mathcal{C} \\ (p, k) &\mapsto c = enc_k(p). \end{aligned}$$

The algorithm dec computes the unique plaintext $p \in \mathcal{P}$ given a valid ciphertext $c \in \mathcal{C}$ and a valid key $k \in \mathcal{K}$:

$$\begin{aligned} dec : \mathcal{C} \times \mathcal{K} &\rightarrow \mathcal{P} \\ (c, k) &\mapsto p = dec_k(c). \end{aligned}$$

The algorithms enc and dec are related by the property that

$$\forall p \in \mathcal{P}\ \forall k \in \mathcal{K} : dec_k(enc_k(p)) = p.$$

A symmetric encryption scheme that takes as input a plaintext block of a fixed size and computes a ciphertext block of fixed length is called *block cipher*. In a so called *iterated block cipher* several transformations are sequentially applied repeatedly.

AES is an iterated block cipher with a fixed block length of 128 bits. The key length can be 128, 192 or 256 bits. Depending on the chosen key length AES is named AES-128, AES-192 or AES-256, respectively. To simplify notation we only describe AES-128. Similar descriptions of the other variants are given in (Daemen and Rijmen 2002) or (NIST 2001).

2.2 Basic Algebraic Structures of AES

The design of AES makes use of several algebraic structures. In this section we briefly describe each of these structures together with their associated operations.

2.2.1 Representation of Data

The basic information unit of AES is a byte consisting of 8 bits. Depending on the underlying algebraic structure AES deals with different representations of bytes. Firstly, a byte b can be written in the *binary notation* as $b = (b_7, \ldots, b_0)$ where each $b_i \in \mathbb{F}_2$. We can also interpret b as a natural number $\sum_{i=0}^{7} b_i \cdot 2^i$ between 0 and 255 and represent it by its *hexadecimal notation* xy where $x, y \in \{0, 1, \ldots, 9, A, B, C, D, E, F\}$. When working in a finite field or ring the *polynomial notation*

$$b = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

with coefficients in $\mathbb{F}_2$ is used.

2.2.2 The Finite Field $\mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x + 1\rangle$

One of the algebraic structures of AES is the finite field with 256 elements. To be more precise, AES uses the polynomial

$$m := x^8 + x^4 + x^3 + x + 1$$

that is irreducible over $\mathbb{F}_2$ to define the finite field

$$\mathbb{F}_{256} = \mathbb{F}_2[x]/\langle m\rangle.$$

The polynomial representation of a byte b is considered as an element of $\mathbb{F}_{256}$. In the sequel, we briefly describe the associated operations.

The addition of two elements of $a, b \in \mathbb{F}_{256}$ is computed as

$$a + b = \sum_{i=0}^{7} (a_i \oplus b_i)x^i$$

where $\oplus$ denotes the addition in $\mathbb{F}_2$.

The multiplication of two elements $a, b \in \mathbb{F}_{256}$ is computed as

$$a \cdot b = \left(\sum_{i=0}^{7} a_i x^i\right) \cdot \left(\sum_{i=0}^{7} b_i x^i\right) \pmod{x^8 + x^4 + x^3 + x + 1}.$$

Obviously, $a = 1 \in \mathbb{F}_{256}$ is the neutral element of the multiplication. Hence, a multiplication by $a = 1$ is the identity. Furthermore, the multiplication by $a = x \in \mathbb{F}_{256}$ can be implemented very efficiently. To do so, the coefficients of b are shifted one position to the left setting the rightmost coefficient to 0:

$$x \cdot b = b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x.$$

To determine the correct reduced result we distinguish two cases: If $b_7 = 0$ then

$$\begin{aligned} & b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \\ = \; & b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \end{aligned}$$

is already in the reduced form and we do not need a further reduction. If $b_7 = 1$ then we have to reduce the result modulo the polynomial m as defined above. We can compute the correct reduced result by simply adding m to the product $x \cdot b$:

$$\begin{array}{cccccccccccccccccc} & x^8 & + & b_6x^7 & + & b_5x^6 & + & b_4x^5 & + & b_3x^4 & + & b_2x^3 & + & b_1x^2 & + & b_0x & + & 0 \\ + & x^8 & & & & & & & + & x^4 & + & x^3 & & & + & x & + & 1 \\ \hline \multicolumn{16}{r}{b_6x^7 + b_5x^6 + (b_4+1)x^5 + (b_3+1)x^4 + (b_2+1)x^3 + b_1x^2 + (b_0+1)x} & + & 1 \end{array}$$

Algorithm 1 shows the computation of $x \cdot b \pmod{m}$ called *xtime*.

Algorithm 1 xtime

Input: $b = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \in \mathbb{F}_2[x]/\langle m \rangle$
Output: $x \cdot b \in \mathbb{F}_2[x]/\langle m \rangle$
$c \leftarrow b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x$ {left shift of coefficients}
if $b_7 = 0$ **then**
    Return c {already correct result}
else
    Return $c + m$ {reduce and return}
end if

Inversion For every element a of the multiplicative group $\mathbb{F}_{256}^\times$ there exists an unique element $b \in \mathbb{F}_{256}^\times$ such that $ab = 1$. The element $b = a^{-1}$ is called the inverse of a. We extend the inversion to all elements of $\mathbb{F}_{256}$ by defining the function

$$\begin{aligned} \mathrm{INV}: \mathbb{F}_{256} &\rightarrow \mathbb{F}_{256} \\ a &\mapsto \begin{cases} a^{-1} & \text{, if } a \in \mathbb{F}_{256}^\times \\ 0 & \text{, if } a = 0 \end{cases} \end{aligned}$$

By Lagrange's Theorem, we can compute $\mathrm{INV}(a)$ in $\mathbb{F}_{256}$ by raising a to the 254th power:

$$\mathrm{INV}(a) = a^{254} \in \mathbb{F}_{256}.$$

We can use the repeated squaring algorithm to compute the power of an element efficiently. See for example (von zur Gathen and Gerhard 2003) or (Shoup 2005) for a comprehensive treatise of the topic.

2.2.3 The Ring $\mathbb{F}_2[x]/\langle x^8 + 1 \rangle$

Another algebraic structure that is used in AES is the ring $\mathbb{F}_2[x]/\langle x^8 + 1 \rangle$. Since

$$x^8 + 1 = (x + 1)^8 \in \mathbb{F}_2[x]$$

is not irreducible over $\mathbb{F}_2$, the ring $\mathbb{F}_2[x]/\langle x^8 + 1 \rangle$ does not form a field and we cannot invert each of its elements $b \neq 0$. The representation of data bytes as an element of the ring is again the polynomial representation like in $\mathbb{F}_{256}$ as described above. Beside computing the reductions modulo $x^8 + 1$, addition and multiplication are defined as above. Hence, for two elements $a, b \in \mathbb{F}_2[x]/\langle x^8 + 1 \rangle$

$$a + b = \sum_{i=0}^{7} (a_i \oplus b_i) x^i$$

and

$$a \cdot b = \left(\sum_{i=0}^{7} a_i x^i \right) \cdot \left(\sum_{i=0}^{7} b_i x^i \right) \pmod{x^8 + 1}.$$

2.2.4 The Ring $\mathcal{R} = \mathbb{F}_{256}[y]/\langle y^4 + 1 \rangle$

AES also deals with 4-tuples of bytes. Here, each byte, considered as an element of $\mathbb{F}_{256}$ as described above, is a coefficient of a polynomial

$$\beta = \beta_3 y^3 + \beta_2 y^2 + \beta_1 y + \beta_0 \pmod{y^4 + 1}$$

of degree less than 4. The polynomials described above form the ring

$$\mathcal{R} := \mathbb{F}_{256}[y]/\langle y^4 + 1 \rangle.$$

For two elements $\alpha = \sum_{i=0}^{3} \alpha_i y^i \in \mathcal{R}$ and $\beta = \sum_{i=0}^{3} \beta_i y^i \in \mathcal{R}$ the sum $\alpha + \beta$ is computed as

$$(\alpha_3 + \beta_3)y^3 + (\alpha_2 + \beta_2)y^2 + (\alpha_1 + \beta_1)y + (\alpha_0 + \beta_0),$$

where the addition of two coefficients is computed in $\mathbb{F}_{256}$.

The product $\alpha \cdot \beta$ is computed as

$$\sum_{i=0}^{3} \alpha_i y^i \cdot \sum_{i=0}^{3} \beta_i y^i \pmod{y^4 + 1}.$$

2.3 The Standard Implementation of AES

After explaining the basic algebraic structures, we now describe the standard implementation of AES as defined in (Daemen and Rijmen 2002) and (NIST 2001). As mentioned above the basic information unit of AES is a byte. 16 bytes arranged in a 4×4 matrix form a so called *state*.

To process a plaintext block p of 128 bits, p is transformed into a state. To do so p is divided into 16 bytes $p_0, p_1, \ldots, p_{15}$. The bytes are mapped to a 4×4 array as shown in Figure 2.1.

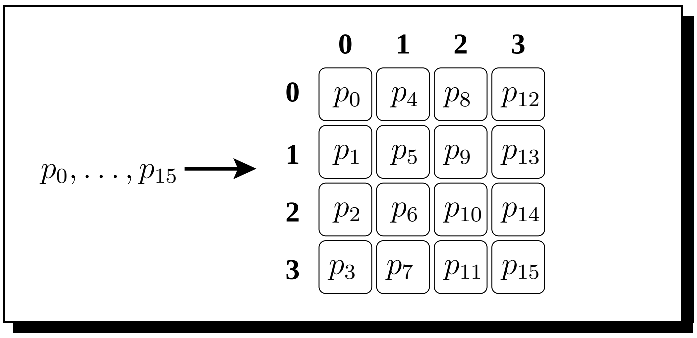

Figure 2.1: Mapping the plaintext p into a state

AES is an iterated block cipher. During the AES encryption several different transformations grouped in so called rounds are repeatedly applied on the state. In the sequel, we first describe each of these transformations and then provide the complete encryption algorithm.

2.3.1 State Transformations

The `SubBytes` (`SB`) Transformation

`SubBytes` is the non-linear transformation of AES.

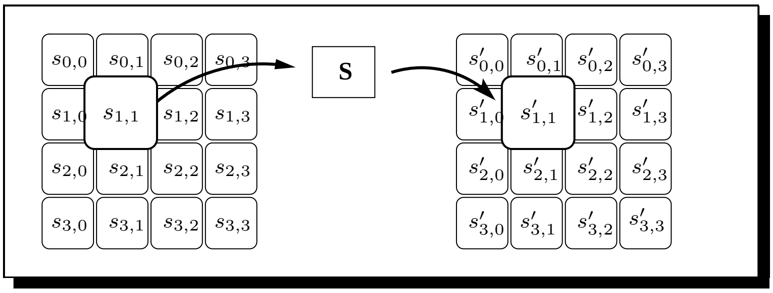

Figure 2.2: The `SubBytes` transformation

It substitutes each byte of the state independently of the other bytes by applying a fixed mapping. In the first step of this mapping each byte b considered as an element of $\mathbb{F}_{256}$ is substituted by its inverse $\text{INV}(b)$. In the second step, the $\text{INV}(b)$ is interpreted as an element of the ring $\mathcal{R}$. A fixed affine mapping in the ring $\mathcal{R}$ is applied to $\text{INV}(b)$:

$$(x^4 + x^3 + x^2 + x + 1) \cdot \text{INV}(b) + (x^6 + x^5 + x + 1) \pmod{x^8 + 1}. \tag{2.1}$$

To apply the mapping efficiently it is usually precomputed for all 256 possible different inputs and the result is stored in a table of size 256 bytes. This table is called the substitution box (*sbox*) **S**. We denote the application of the mapping to a byte b by $\mathbf{S}[b]$. Figure 2.2 depicts the application of the sbox.

The `ShiftRows` (`SR`) Transformation

The `ShiftRows` transformation performs a cyclic shift to each row of the state. Each row is shifted by a fixed byte positions to the left. The first row is not shifted, the second row is shifted one position to the left, the third row is shifted two positions to the left and the fourth row is shifted three positions to the left. The `ShiftRows` operation is depicted in Figure 2.3.

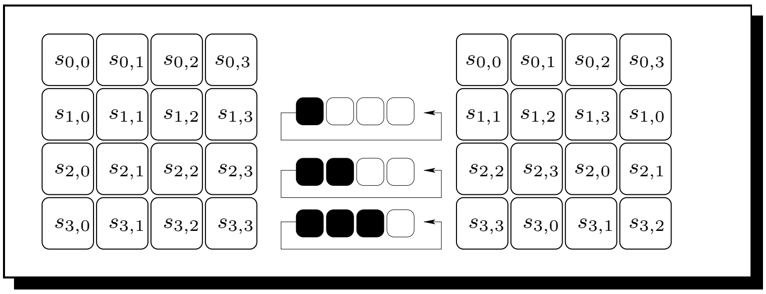

Figure 2.3: The `ShiftRows` transformation

The `MixColumns` (`MC`) Transformation

The `MixColumns` transformation performs a linear combination of the bytes of a column. Each byte of the state is interpreted as an element of $\mathbb{F}_{256}$. The four bytes $\beta_0, \beta_1, \beta_2, \beta_3$ of a

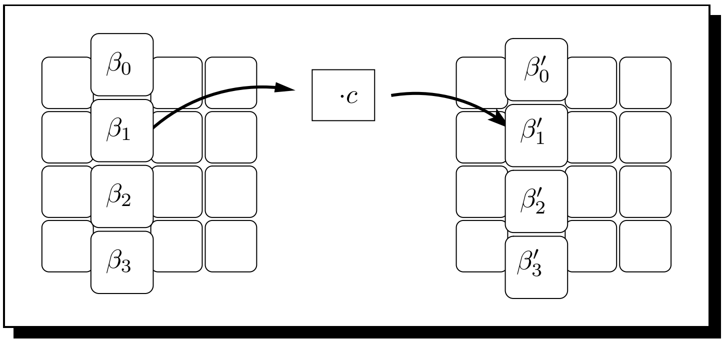

Figure 2.4: The `MixColumns` transformation

column are considered as the coefficients of a polynomial

$$\beta = \beta_3 y^3 + \beta_2 y^2 + \beta_1 y + \beta_0 \in \mathcal{R}$$

of degree less than 4 over the ring $\mathcal{R} = \mathbb{F}_{256}[y]/\langle y^4 + 1 \rangle$. The polynomial β is then multiplied with a fixed polynomial:

$$c := 03y^3 + 01y^2 + 01y + 02 \in \mathbb{F}_{256}[y]/\langle y^4 + 1 \rangle.$$

`MixColumns` is depicted in Figure 2.4. Alternatively, we can represent the `MixColumns` transformation as a matrix multiplication:

$$\underbrace{\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}}_{\in \mathbb{F}_{256}^{4\times 4}} \cdot \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \begin{bmatrix} \beta_0' \\ \beta_1' \\ \beta_2' \\ \beta_3' \end{bmatrix}$$

The AddRoundKey Transformation

To introduce the secret key into the encryption, the `AddRoundKey` transformation is used. The so called *key schedule* is explained in Section 2.3.3 and gets as input the cipher key and generates a so called roundkey for every round of AES. The round key is of the same size as the encryption state, i.e., it forms a 4×4 byte matrix. The `AddRoundKey` transformation combines a byte b of the state with its corresponding byte k of the round key by computing the bitwise addition modulo 2 (XOR): $b \oplus k$. The `AddRoundKey` transformation is depicted in Figure 2.5.

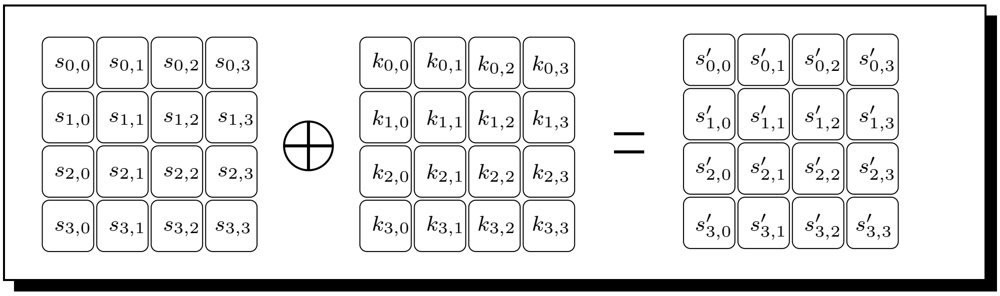

Figure 2.5: The `AddRoundKey` transformation

2.3.2 Encryption

The AES encryption entirely consists of the four state transformations. A round of the AES encryption is composed by consecutively applying the state transformations to the state in the order shown in Algorithm 2. The complete encryption algorithm is shown in Algorithm 3.

Algorithm 2 A round of the AES encryption

```
1: SubBytes
2: ShiftRows
3: MixColumns
4: AddRoundKey
```

It consists of an initial `AddRoundKey` and 9 times applying the AES round as described in Algorithm 2. After that a truncated round is applied that only consists of `SubBytes`, `ShiftRows` and `AddRoundKey`.

2.3.3 Key Expansion

AES-128 applies the `AddRoundKey` transformation eleven times on the intermediate state. `AddRoundKey` is applied before the first round, in each of the nine rounds and in the truncated

Algorithm 3 Complete AES encryption

Input: plaintext $p_0, \ldots, p_{15} \in \{0,1\}^8$, key k
Output: ciphertext $c_0, \ldots, c_{15} \in \{0,1\}^8$

```
AddRoundKey
for i = 1 to 9 do
  SubBytes
  ShiftRows
  MixColumns
  AddRoundKey
end for
SubBytes
ShiftRows
AddRoundKey
```

last round. To generate different round keys for each of these applications of `AddRoundKey` a so called *expanded key* w is derived from the cipher key $k = k_0, \ldots, k_{15} \in \left(\{0,1\}^8\right)^{16}$ as follows. The cipher key k is mapped to a 4×4 state matrix similar to the mapping of a plaintext to a state as shown in Figure 2.1 (page 9). The four bytes of each column of this matrix form a so called *word*. We define two operations on words. The first operation is the so called `SubWord` operation. `SubWord` applies the sbox to every byte of the word:

$$\begin{aligned} \texttt{SubWord}: \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 &\rightarrow \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 \\ (\beta_0, \beta_1, \beta_2, \beta_3) &\mapsto (\mathbf{S}[\beta_0], \mathbf{S}[\beta_1], \mathbf{S}[\beta_2], \mathbf{S}[\beta_3]). \end{aligned}$$

The second operation is `RotWord` that cyclically shifts the 4 bytes of a word one postion to the left.

$$\begin{aligned} \texttt{RotWord}: \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 &\rightarrow \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 \\ (\beta_0, \beta_1, \beta_2, \beta_3) &\mapsto (\beta_1, \beta_2, \beta_3, \beta_0). \end{aligned}$$

Furthermore, for $i \geq 1$ let

$$\mathrm{Rcon}[i] := (x^{i-1}, 0, 0, 0) \in (\mathbb{F}_2[x]/\langle m \rangle)^4$$

the so called *round constant* for the ith round key. The expanded key w is then computed according to Algorithm 4.

The round key for round i is extracted from the expanded key w by mapping the words $w_{4i}, \ldots, w_{4i+3}$ of the expanded key w to the columns of a 4×4 byte matrix.

Algorithm 4 Key schedule of AES-128 in pseudocode

Input: cipherkey $k = k_0, \ldots, k_{15} \in \{0,1\}^8$
Output: expanded key $w = w_0, \ldots, w_{43} \in \{0,1\}^{32}$

```
for i ← 0, ..., 3 do
    w_i = (k_{4·i}, k_{4·i+1}, k_{4·i+2}, k_{4·i+3});
end for
for i ← 4, ..., 43 do
    temp = w_{i−1}
    if (i ≡ 0 mod 4) then
        temp = SubWord(RotWord(temp)) ⊕ Rcon[i/4]
    end if
    w_i = w_{i−4} ⊕ temp;
end for
```

2.3.4 Decryption

The decryption of AES, that is determining the unique plaintext given the corresponding ciphertext and the correct secret key, is done by reverting every transformation that was applied in the encryption. In the sequel, we show how every single transformation can be inverted. Hence, applying the inverse of each transformation in the reversed order will compute the correct plaintext.

The `InvSubBytes` Transformation

To undo the `SubBytes` transformation that substituted a byte b with

$$(x^4 + x^3 + x^2 + x + 1) \cdot \mathrm{INV}(b) + (x^6 + x^5 + x + 1) \pmod{x^8 + 1}$$

we proceed in two steps. Firstly, notice that the function INV is self inverse. Secondly, the affine mapping in the ring $\mathcal{R}$ is invertible having the inverse

$$(x^6 + x^3 + x) \cdot b + (x^2 + 1) \pmod{x^8 + 1}.$$

Hence, the inverse transformation `InvSubBytes` of the `SubBytes` transformation is given by applying the mapping

$$\mathrm{INV}\left((x^6 + x^3 + x) \cdot b + (x^2 + 1)\right) \pmod{x^8 + 1} \tag{2.2}$$

to every byte of the state.

To increase the efficiency one can precompute all 256 possible values and store them in a table called the *inverse sbox* $\mathbf{S}^{-1}$.

The `InvShiftRows` Transformation

The `ShiftRows` transformation is obviously invertible by cyclically shifting the bytes of a row by the appropriate number of position to the right. I.e., shifting the second row one position, the third row two positions and the fourth row three positions to the right cancels the effect of `ShiftRows` on a state.

The `InvMixColumns` Transformation

In the `MixColumns` transformation each column of the state is interpreted as an element of the ring $\mathbb{F}_{256}[y]/\langle y^4+1\rangle$ is multiplied by a fixed polynomial $c = 03 \cdot y^3 + 01 \cdot y^2 + 01 \cdot y + 02$. Since $\gcd(c, y^4+1) = 1$ the inverse of c exists:

$$c^{-1} := 0\mathrm{B} \cdot y^3 + 0\mathrm{D} \cdot y^2 + 09 \cdot y + 0\mathrm{E} \in \mathbb{F}_{256}[y]/\langle y^4+1\rangle.$$

Multiplying each row interpreted as an element of $\mathbb{F}_{256}[y]/\langle y^4+1\rangle$ with c^{-1} cancels the effect of the `MixColumns` operation on a state.

The `InvAddRoundKey` Transformation

The round key is combined with the state by bitwise adding (XOR) the bytes of the round key with the corresponding bytes of the state. Since the XOR operation is its own inverse adding the round key again cancels the effect of the `AddRoundKey` transformation.

After specifying the inverse of each individual transformation we can compute the decryption of a ciphertext by applying the inverse transformations in the reversed order as shown in Algorithm 5.

Algorithm 5 Complete AES decryption

Input: ciphertext $c_0, \ldots, c_{15} \in \{0,1\}^8$, key k
Output: plaintext $p_0, \ldots, p_{15} \in \{0,1\}^8$

```
InvAddRoundKey
InvShiftRows
InvSubBytes
for i = 9 to 1 do
  InvAddRoundKey
  InvMixColumns
  InvShiftRows
  InvSubBytes
end for
AddRoundKey
```

2.4 The Fast Implementation of AES

Combining the transformations `SubBytes`, `ShiftRows` and `MixColumns` as described in Section 4.2 of (Daemen and Rijmen 2002) leads to an alternative description of AES. Notice that the operations `SubBytes` and `ShiftRows` can be exchanged. `SubBytes` substitutes the bytes independent of their position whereas `ShiftRows` changes the position of the bytes independent of their values.

Let

$$s := \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

be the state before it enters an encryption round. For $0 \leq j \leq 3$ consider the four bytes $s_{0,j}, s_{1,j+1}, s_{2,j+2}, s_{3,j+3}$ of the state s where the indices are computed modulo 4. The four bytes are transformed by `SubBytes` and `ShiftRows` such that they form the new jth column:

$$\begin{bmatrix} \mathbf{S}[s_{0,j}] \\ \mathbf{S}[s_{1,j+1}] \\ \mathbf{S}[a_{2,j+2}] \\ \mathbf{S}[a_{3,j+3}] \end{bmatrix}.$$

The application of `MixColumns` and `AddRoundKey` leads to

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{S}[s_{0,j}] \\ \mathbf{S}[s_{1,j+1}] \\ \mathbf{S}[s_{2,j+2}] \\ \mathbf{S}[s_{3,j+3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

We rewrite the matrix multiplication as the linear combination of the column vectors

$$\mathbf{S}[s_{0,j}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus \mathbf{S}[s_{1,j+1}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus \mathbf{S}[s_{2,j+2}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus \mathbf{S}[s_{3,j+3}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Based on this linear combination we can construct new sboxes

$$\mathbf{T}_0, \mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3 \ : \ \{0,1\}^8 \rightarrow \left(\{0,1\}^8\right)^4$$

as follows:

$$\mathbf{T}_0[a] := \begin{bmatrix} \mathbf{S}[a] \cdot 02 \\ \mathbf{S}[a] \cdot 01 \\ \mathbf{S}[a] \cdot 01 \\ \mathbf{S}[a] \cdot 03 \end{bmatrix}, \mathbf{T}_1[a] := \begin{bmatrix} \mathbf{S}[a] \cdot 03 \\ \mathbf{S}[a] \cdot 02 \\ \mathbf{S}[a] \cdot 01 \\ \mathbf{S}[a] \cdot 01 \end{bmatrix}, \mathbf{T}_2[a] := \begin{bmatrix} \mathbf{S}[a] \cdot 01 \\ \mathbf{S}[a] \cdot 03 \\ \mathbf{S}[a] \cdot 02 \\ \mathbf{S}[a] \cdot 01 \end{bmatrix}, \mathbf{T}_3[a] := \begin{bmatrix} \mathbf{S}[a] \cdot 01 \\ \mathbf{S}[a] \cdot 01 \\ \mathbf{S}[a] \cdot 03 \\ \mathbf{S}[a] \cdot 02 \end{bmatrix}.$$

Each of the sboxes $\mathbf{T}_0, \mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3$ has 256 entries of size four bytes. 4 bytes can be encrypted one (full) round by computing

$$\mathbf{T}_0[a_{0,j}] \oplus \mathbf{T}_1[a_{1,j+1}] \oplus \mathbf{T}_2[a_{2,j+2}] \oplus \mathbf{T}_3[a_{3,j+3}] \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}.$$

For the last (truncated) round that does not have a `MixColumns` transformation things are more simple. We could simply apply the standard sbox $\mathbf{S}$ to every byte of the state. However, to increase the efficiency on 32 bit platforms (Daemen and Rijmen 2002) suggested to use the sbox

$$\begin{aligned} \mathbf{T}_4 \;:\; \{0,1\}^8 &\rightarrow \left(\{0,1\}^8\right)^4 \\ a &\mapsto \mathbf{S}[a], \mathbf{S}[a], \mathbf{S}[a], \mathbf{S}[a]. \end{aligned}$$

Merging the transformations as described above leads to a description of AES that only uses applications of the sboxes and key additions to compute the correct AES encryption.

Chapter 3

Security and Side Channel Attacks

Classical cryptography covers several different security notions, e.g., security against known plaintext attacks or chosen plaintext attacks. But all the different security notions share at least one assumption: The encryption function is a black box. I.e., the only information an attacker can get or influence is the plaintext and the ciphertext of the encryption function as depicted in Figure 3.1. Here Alice and Bob want to communicate confidentially over an insecure channel. To protect their communication they encrypt the messages in a private environment before sending them. The attacker named Eve wants to obtain information about the messages or the key used for encryption.

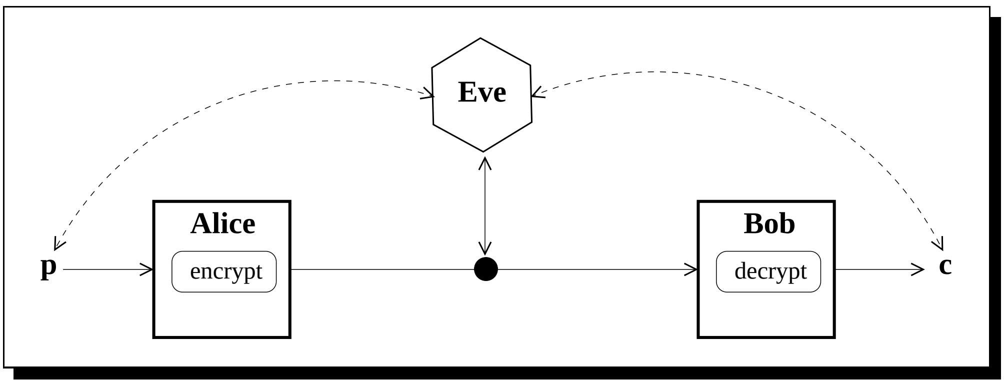

Figure 3.1: Black box model of classical cryptography

However, cryptographic algorithms have to be implemented either in hardware or software. It turned out that implementations of cryptographic algorithms leak some information about the cryptographic operations through so called *side channels*. The information that leaks is called *side channel information*, e.g., the time it takes to encrypt a plaintext or the power consumption etc. Side channel information depends on the implementation and its inputs, i.e., the plaintext and the secret key. An attack that uses side channel information is called *side channel attack* (SCA). It turns out that side channel attacks are much more efficient than classical attacks for virtually every cryptographic algorithm. Hence, to analyze the security of cryptographic algorithms it is essential that side channels are considered as a real threat and are incorporated into the black box model. This leads to an extended black box model like the one depicted in Figure 3.2. However, securing algorithms against side channel attacks

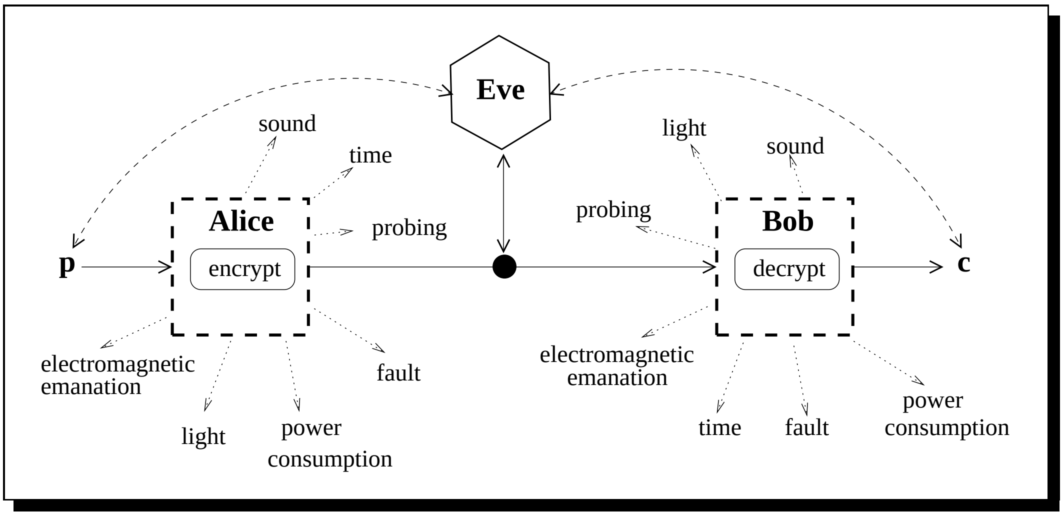

Figure 3.2: Extended black box model that incorporates side channels

is quite tricky. At least two problems occur:

1. It is unclear how to determine all side channels. So far, no security model that considers all side channels is known.

2. It is difficult to prevent the leakage of information. Most of the countermeasures proposed so far only thwart a certain way of exploiting side channel information.

We introduce a model for analyzing the security against side channel attacks in Chapter 4.

3.1 General Principles of Side Channel Attacks

In the sequel, we describe the general principle of side channel attacks and the assumptions that are necessary for mounting a side channel attack on an implementation. The essential assumptions concerning an attacker $\mathcal{A}$ that exploits side channel information are:

Assumption 1 (Kerckhoffs' extended principle) *$\mathcal{A}$ knows all technical details about the underlying cryptographic algorithm and its implementation.*

This assumption is implicitely used in all side channel attacks. In the following we simply refer to it as *Kerckhoffs' extended principle*.

Assumption 2 *$\mathcal{A}$ is able to get plaintexts (or ciphertexts) of encryptions. Furthermore, for each encryption $\mathcal{A}$ is able to obtain side channel information.*

The general structure of a side channel attack consists of the following steps:

measurement step In the measurement step the adversary $\mathcal{A}$ obtains the side channel information of the implementation together with the corresponding plaintext and/or ciphertext. To perform the measurement step, $\mathcal{A}$ needs access to the implementation of the algorithm. Therefore this step is also called online step.

analysis step $\mathcal{A}$ interprets the information collected in the measurement step and tries to connect the side channel information to some property of an intermediate state of the encryption. This analysis lets $\mathcal{A}$ derive some information about the secret key. Depending on the side channel attack the analysis step can determine the secret key uniquely or reduces the number of key candidates significantly such that a brute force attack is applicable. The analysis can be performed without access to the implementation and hence this step is also called offline step.

3.2 Side Channels

In the sequel we give an overview over the most commonly analyzed side channels and specify the common structure of side channel attacks.

3.2.1 Timing Attack

The first publication of a successful timing attack was Kochers timing attack on modular exponentiation as used in RSA (Kocher 1996). In the asymmetric cipher RSA, a ciphertext c viewed as an element of the multiplicative group $\mathbb{Z}_N^*$ is decrypted by raising it to the d-th power

$$p = c^d \bmod N,$$

where $N \in \mathbb{Z}$ is the public modulus and $d \in \mathbb{Z}_{\varphi(N)}^*$ is the secret exponent. The exponentiation can be computed efficiently using the repeated squaring algorithm or a variation of it. Kocher showed how to determine d efficiently by analyzing time measurements of decryptions of many different ciphertexts.

Quisquater's Timing Attack on RSA

(Dhem, Koeune, Leroux, Mestré, Quisquater and Willems 1998) improved the timing attack on RSA that uses a fast modular multiplication method called Montgomery multiplication (Montgomery 1985). The structure of the attack is as follows. Let $[d_0, d_1, \ldots, d_n]$ be the binary representation of d. Knowing the bits $d_0, \ldots, d_{i-1}$ of d, the bit d_i can be determined by computing the following steps for many ciphertexts c:

1. measure the running time T of the decryption of c
2. compute $z = c^{[d_0,d_1,\dot{;}d_{i-1},0]}$
3. if computing $z \cdot c$ takes "long" then put T into set S_1
4. else put T into set S_2

After that, the attacker $\mathcal{A}$ compares the average timings of S_1 and S_2. If they differ significantly, $\mathcal{A}$ assumes that $d_i = 1$. Otherwise he assumes that $d_i = 0$.

Before starting the attack, $\mathcal{A}$ first implicitly assumes that $d_i = 1$ which implies that a modular multiplication is computed in step i of the decryption. Since $\mathcal{A}$ knows all preceeding bits, he can compute the intermediate result of the decryption right before step i. He splits the set of time measurements depending on the time it would take to compute the multiplication in step i. The set S_1 stores all timing measurements of ciphertexts that would take a "long" time for the multiplication. The set S_2 stores all timing measurements of ciphertexts that would take a "short" time for the multiplication. The assumption is that if the multiplication takes a long time than it is more likely that the overall encryption time is greater than the encryption time of a ciphertext for which the multiplication takes a short time. Hence, if $d_i = 1$ then the average running time of set S_1 should be significantly larger than the average running time of the set S_2. On the other hand, if $d_i = 0$ then no modular multiplication will be computed. The splitting of measurements into the two sets is assumed to be random and we expect that the average running times do not differ significantly.

There are several different variant of timing attacks. (Schindler 2000) adapted the concept of timing attacks to RSA using the Chinese Remainder Theorem. (Cathalo, Koeune and Quisquater 2003) developed a different type of timing attack to break the identification scheme GPS of (Baudron, Boudot, Bourel, Bresson, Corbel, Frisch, Gilbert, Girault, Goubin, Misarsky, Nguyen, Patarin, Pointcheval, Stern, Traor and Poupard 2000). Symmetric ciphers are also susceptible to timing attacks. (Hevia and Kiwi 1999) showed how to determine the secret DES key and (Koeune and Quisquater 1999) obtained secret AES keys by mounting timing attacks.

The power of timing attacks goes far beyond local attacks. (Brumley and Boneh 2005) demonstrated that remote timing attacks are possible. They determined the secret RSA exponent of a web server running openssl by remotely taking time measurements over a computer network. This remote timing attack was improved by (Acıiçmez, Schindler and Koç 2005).

3.2.2 Power Analysis

The idea of power analysis is that the power consumption of a cryptographic device is related to intermediate results of an encryption algorithm and hence depends on the secret key. The first successful power attacks are due to (Kocher, Jaffe and Jun 1998). Power analysis can be divided into simple power analysis (SPA) and differential power analysis (DPA). In an SPA, the attacker analyzes a single power trace to figure out which operation and operands were executed at what time.

As the name suggests, differential power analysis is based on the differences of power traces obtained from many different inputs. Similar to timing attacks, the attacker $\mathcal{A}$ splits a large set of power traces into two sets depending on some guesses of parts of the key. For each plaintext the attacker does the following. If the guess of the part of the key implies that a certain operation during the encryption should consume a lot of power then the obtained power trace is put into set S_1. On the other hand, if the key guess implies that the operation does not consume much power the trace is put into set S_2. In the end, the attacker computes the difference of the average traces of both sets. If there is a peak in the difference trace than the attacker assumes that the guess of the part of the key was correct. Otherwise he assumes that the guess was wrong.

The underlying idea is similar to the one of the timing attack. If the key guess is correct than all power traces in the set S_1 show a high power consumption (peak) at the time when the certain operation is executed. Hence, the average power trace of set S_1 also shows this peak. The average trace of set S_2 does not have this peak. Therefore, the peak of S_1 will be visible in the difference of the two average traces.

If the key guess was wrong than the attacker wrongly decides whether the operation would consume a lot of power or not. The assumption is that in this case the assignment of power traces into the sets S_1, S_2 is random. Hence, we expect that when computing the average traces the peaks in the power traces cancel out and we get a smooth difference trace.

3.2.3 Fault Attacks

The main idea of fault attacks is to obtain information about the secret key by inducing faults into the cryptographic operation. We deal with fault attacks in more detail in Chapter 5 (page 49).

3.2.4 Cache Attacks

A cache is a fast buffer memory that can be accessed faster than the main memory. Hence, buffering data that is used more often in the cache increases the performance of a computer. In a cache attack the attacker observes information about the cache behavior of an algorithm. E.g., he figures out how many cache accesses happened or which operation caused a cache access. We analyze cache attacks in more detail in Chapter 6 (page 71).

3.2.5 Other Side Channel Attacks

Beside the side channels described above there are several other ways for an attacker to obtain information about the internal states of an cryptography algorithm. (van Eck 1985) shows how to reconstruct the content of a computer display by analyzing the electromagnetic radiation of the monitor. Neal Stephenson treats the so called *van Eck phreaking* of attack in his novel "Cryptonomicon" (Stephenson 1999). The concept of using electromagnetic radiation to attack cryptographic algorithms was demonstrated in (Quisquater and Samyde 2001), (Gandolfi, Mourtel and Olivier 2001) and (Kuhn 2003).

Another example for a side channel attack proposed in (Shamir and Tromer 2004) is to analyze the sound a computer generates while operating with the secret key. Further kinds of side channel attacks are among others so called frequency based attacks (Tiu 2005), visible light attacks (Kuhn 2002) and scan based attacks (Yang, Wu and Karri 2004).

3.3 Countermeasures

In general, there are two strategies to thwart side channel attacks. The first strategy is to prevent the information leakage. E.g., to thwart timing attacks one could build an implementation that uses constant execution time for all possible inputs. However, this approach has several disadvantages. Firstly, building such an implementation is costly because it has to consider all details of the underlying hardware and other parts of the environment. Secondly, missing one of the details could lead to an implementation that is susceptible to other side channel attacks. The third disadvantage of this approach is that it leads to inefficient implementations that have to be redesigned for every different environment.

The second strategy is to randomize the intermediate values of an implementation such that the leaking information is useless for an attacker. Furthermore, the implementation has to ensure that the correct ciphertext is computed in the end. Of course, this approach needs random values for obfuscating intermediate values. But randomization has several advantages over the strategy of preventing information leakage. The first advantage is that one can define a general model to analyze the effectiveness of the randomization. Furthermore, the randomization can be done independently of the underlying hardware. Therefore one can reuse randomized algorithms on several different platforms.

In the next chapter, we will present such a randomization strategy to provably protect the AES against side channel attacks in a strong model.

Chapter 4

Provably Secure Randomization of Cryptographic Algorithms

The security of AES against Simple Power Analysis (SPA), Differential Power Analysis (DPA), Higher Order Differential Power Analysis (HODPA) as published in (Kocher et al. 1998), (Kocher et al. 1999), and Timing Attacks (Kocher 1996) has received considerable attention since the beginning of the AES selection process. (Koeune and Quisquater 1999) describe timing attacks against careless implementations of AES. (Biham and Shamir 1999) and (Daemen and Rijmen 1999) discuss DPA attacks on the AES candidates in software based solutions. (Örs, Gürkaynak, Oswald and Preneel 2004) describe the first power analysis-based attack on a dedicated ASIC implementation of AES and (Mangard 2002) discusses an SPA attack on the key schedule of AES.

As a result of these attacks, numerous hardware and algorithmic countermeasures have been proposed. Hardware methodologies were proposed right from the beginning including randomized clocks, memory encryption schemes, see (Clavier, Coron and Dabbous 2000) and (Golić 2003), power consumption randomization (Daemen and Rijmen 1999), and decorrelating the external power supply from the internal power consumed by the chip. Moreover, the use of different hardware logic, such as complementary logic (Daemen and Rijmen 1999), sense amplifier based logic (SABL) and asynchronous logic (Fournier, Moore, Li, Mullins and Taylor 2003) and (Moore, Anderson, Mullins, Taylor and Fournier 2003) has also been proposed. Some of these methods soon proved to be ineffective while other more successful countermeasures are very costly in terms of development, area and power consumption. For example, the techniques in (Daemen and Rijmen 1999), (Tiri, Akmal and Verbauwhede 2002), (Tiri and Verbauwhede 2003), (Fournier et al. 2003) and (Moore et al. 2003) require about twice as much area and will consume twice as much power as an implementation that is not protected against power attacks. In addition, hardware countermeasure will only protect against known techniques and attacks. They cannot provide security in a precisely defined mathematical sense. Hence, although hardware countermeasures are an important defense against side channel attacks, they should be complemented by algorithmic countermeasures that are provably secure in a mathematically precise sense.

In this chapter, we focus on algorithmic countermeasures against timing and power attacks on AES. In general, efficient algorithmic countermeasures against timing and power attacks are based on randomization techniques. Here the problem is to guarantee that all information that is accessible via side channels is random and hence useless to the attacker. Moreover, the randomization must be used in such a way that, at the end of the algorithm, the correct encryption or signature corresponding to the input plaintext is obtained. Randomized algorithmic countermeasures against timing and power attacks include secret-sharing schemes, independently proposed by (Goubin and Patarin 1999) and (Chari, Jutla, Rao and Rohatgi 1999) as well as methods based on the idea of masking all data and intermediate results during an encryption operation, originally introduced by (Messerges 2000). This chapter is organized as follows.

Section 4.1: Security Model 28

In this section we introduce and discuss our mathematically precise security notion in which we discuss randomization techniques. For our security notion we only make some inevitable assumptions: Firstly, we assume that some (small) part of the computation runs in a protected environment. Secondly, we limit the number of intermediate results that an adversary has access to. Note that previous methods made at least these assumptions. On the other hand, we assume that arbitrary differences in the distribution of an intermediate result that depends on the plaintext or secret key of the cryptosystem can be used to break the system completely. Accordingly, our security notion requires that the distribution of any intermediate result is stochastically independent of the secret key being used and independent of the plaintext. Independent of our research, Golić briefly sketched a similar requirement in (Golić 2003). In the sequel, we call an algorithm *order-d perfectly masked* if the joint distribution of any d intermediate results is independent of the secret key and the plaintext. This notion of security strengthens the security notion proposed in (Chari et al. 1999). Their security notion only requires that the distribution of some side channel information about an intermediate result has to be indistinguishable by an adversary. Since our security notion assumes that even tiny differences in the distribution of the values of intermediate results completely break an implementation of a cryptosystem, this notion is strong and often unrealistic. On the other hand, we will argue that our security notion implies security against most side channel attacks.

Section 4.2: Masking AES 31

In this section we briefly describe the masking techniques proposed so far. The first algorithmic countermeasure against power attacks customized for the AES was the *Transform Masking Method* by (Akkar and Giraud 2001). This method was further simplified by (Trichina, Seta and Germani 2002). It was noticed in (Trichina et al. 2002), (Golić and Tymen 2002) and (Akkar and Goubin 2003) that the multiplicative masking introduced in (Akkar and Giraud 2001) masks *only* non-zero values, i.e., a zero byte will not get masked because of the multiplicative nature of the mask. This feature renders the method of Akkar and Giraud vulnerable to DPAs. A second masking technique for AES is the *Random Representation Method* of (Golić and Tymen 2002). Similar to Akkar and Giraud, Golić and Tymen do not try to show that their technique randomizes all intermediate results. Instead, the authors argue experimentally that using their methods the Hamming weights of all intermediate results are distributed in roughly the same way, independent of the plaintext and the secret key. We conclude that so far customized randomization techniques for AES were based on empirical assumptions about the power of potential adversaries. Then these assumptions were used to define some ad-hoc-model in which to analyze and argue the security of the methods. We believe that this is a potentially dangerous approach.

Section 4.3: Perfectly Masking AES against Order-1 Adversaries 33

Based on our security notion we develop an order-1 perfectly masked algorithm for AES. Hence, this algorithm is secure against any adversary that gets plaintext/ciphertext pairs and a *single* arbitrary intermediate result for each of those pairs. The main problem here is to describe a secure algorithm for the inversion operation that is the main ingredient of the AES `SubBytes` transformation. Our solution is based on a general technique to turn an arbitrary algorithm using arithmetic operations defined over some finite field into a randomized algorithm that securely computes the same function.

Section 4.4: Implementation and Costs 38

We show that masking countermeasures are inexpensive to implement in hardware. Our method amounts to only a 20% increase in the overall area required for an AES hardware implementation when compared to dual-rail logic type countermeasures. To show this, we provide a detailed cost comparison of the different methods. Because our method is based on the usage of multipliers and adders over any binary field, designers might use this method to implement DPA-safe circuits which utilize previously designed multiplier and adder blocks. Moreover, the method is modular and encourages reusability.

Section 4.5: Order-d Perfectly Masking 41

In this section we generalize our method of order-1 perfectly masked algorithms. We show how to design *order-d* perfectly masked algorithms that are secure against adversaries that get the values of a fixed number d of intermediate results.

Section 4.6: Conclusion 47

We conclude the chapter by giving a brief survey of our contribution in the area of building reliable security models and developing provably secure algorithms.

4.1 Security Model

In this section we describe our model which we will use in the sequel to analyze the security of algorithms against side channel attacks. We specify the underlying assumptions that characterize the model.

Let $\mathcal{P}$, $\mathcal{K}$ and $\mathcal{C}$ denote the set of plaintexts, the set of keys and the set of ciphertexts respectively. We consider some encryption function

$$\begin{aligned} \mathrm{enc} : \mathcal{P} \times \mathcal{K} &\rightarrow \mathcal{C} \\ (x, k) &\mapsto c. \end{aligned}$$

Given an algorithm E that evaluates the function enc, for each plaintext $x \in \mathcal{P}$ and key $k \in \mathcal{K}$, we view the computation of $\mathrm{E}(x, k)$ as a sequence of $t \in \mathbb{N}$ *intermediate results*

$$I_1(x, k, R), \ldots, I_t(x, k, R).$$

Each intermediate result I_i may depend on the plaintext x, on the secret key k, and some $R \in \{0, 1\}^\alpha$ for an appropriate constant $\alpha \in \mathbb{N}$. The element R is used to randomize the computation and is chosen uniformly at random from $\{0, 1\}^\alpha$. For simplicity we assume that we have a true random number generator (TRNG) and that the adversary is not able to manipulate the random bits. Note that the ciphertext $\mathrm{enc}(x, k) = I_t(x, k, R)$ only depends on x and k and not on R.

We consider an adversary $\mathcal{A}$ that wants to derive information about the secret key k by using side channel information. To characterize the security model we make the following assumptions:

Assumption 3

1. *The adversary $\mathcal{A}$ can choose an arbitrary number of plaintexts (or ciphertexts) and obtains the corresponding ciphertexts (or plaintexts).*
2. *For each encryption (or decryption), $\mathcal{A}$ gets the values of a constant number d of intermediate results.*

In point 1 of Assumption 3 we allow the adversary to obtain an arbitrary number of (adaptively) chosen plaintext/ciphertext pairs $(x, \mathrm{enc}(x, k))$. Furthermore, for each pair, the adversary $\mathcal{A}$ obtains the values of d intermediate results of his choice. $\mathcal{A}$ may get different intermediate results for different plaintext/ciphertext pairs. The larger the number d of known intermediate results is, the more powerful is $\mathcal{A}$. We call an adversary $\mathcal{A}$, that can get at most d intermediate results for each pair $(x, \mathrm{enc}(x))$ an *order-d adversary*.

So far we considered intermediate results without specifying the possible intermediate results that an adversary may get. We consider an algorithm as a sequence of operations that are treated as encapsulated modules. This leads to a classification of intermediate results into different levels down to the bit level:

1. *Text level:* The whole algorithm is treated as a module. This level is the one of classical cryptography. The only information available to the adversary is the plaintext and the ciphertext.
2. *Block level:* Each part or subroutine of the algorithm is treated as a module. In the case of a block cipher such as the AES, each transformation within a round is treated as a module (`SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`).
3. *Unit level:* Each arithmetic operation is treated as a module. These operations work on the atomic units of information in the cipher. For example, the AES units of information are bytes; no operation acts on single bits or nibbles directly. In hardware terms this level is based on the contents of registers.
4. *Bit level:* Each bit manipulation is treated as a module, for example XOR, shift etc.

Every output of such a module is an *intermediate result*. In this section we concentrate on intermediate results at the unit level. For AES this seems to be a natural choice since basically all operations in AES are arithmetic operations on bytes. Therefore timing, power and fault attacks on AES have focused on these operations as well.

Assumption 4 *Some of the operations of the algorithm E that evaluates the encryption function are protected against $\mathcal{A}$.*

This assumption is inevitable to achieve a reasonable notion of security. To see this, note that the secret key k itself can be considered as an intermediate result. Letting $\mathcal{A}$ obtain k directly would render all algorithms and countermeasures insecure. Hence, we must assume that some parts of the computation run in a guaranteed secure environment. I.e., some intermediate results cannot be accessed by an adversary. At least implicitly, all previously proposed countermeasures against side channel attacks have made the same assumption. Note that modern smartcards are protected by different types of countermeasures like sensors and shields. Hence, the assumption that at least some computations are done in a secure environment is realistic. However, it is desireable to clearly specify and to limit the number of those operations because their protection is expensive.

Assumption 5 *If the joint distribution of d intermediate results depends on the plaintext x and on the secret key k then $\mathcal{A}$ can determine k.*

This assumption strengthens the adversary. If the joint distribution of d intermediate results depends on the secret key then it provides $\mathcal{A}$ some information about k. To simplify and strengthen our security model we assume that in this case $\mathcal{A}$ can determine the entire key k.

Intuitively, we say that the algorithm computing enc is insecure if the joint distribution of the intermediate results that are accessible for an adversary depends on the plaintext x and on the secret key k. To formalize this, fix some d-tuple $I_1, \ldots, I_d$ of intermediate results. For a pair (x, k) of plaintext and key we denote by $D_{x,k}(R)$ the joint distribution of $I_1, \ldots, I_d$ induced by choosing R uniformly at random in $\{0,1\}^\alpha$ for an appropriate constant α. Now we can define our notion of security called perfect masking:

Definition 2 (perfect masking) *An algorithm that evaluates an encryption function enc is order-d perfectly masked if for all d-tuples $I_1, \ldots, I_d$ of intermediate results we have that*

$$D_{x,k}(R) = D_{x',k'}(R) \quad \textit{for all pairs } (x,k), (x',k').$$

For $d = 1$ we say that an algorithm is perfectly masked.

4.1.1 Discussion of the Security Notion

Our notion of security is very strong. Basically, we assume that an adversary can determine the secret key even from tiny differences in the (joint) distribution of intermediate results. In many realistic cases this may not be true. However, we do not want to base our security model on assumptions about technical abilities or limitations adversaries currently have. Instead we want to provide a precise mathematical notion that captures security against current side channel attacks as well as future ones. Our notion of security strengthens the security notion of (Chari et al. 1999). We require that for any two pairs $(x, k), (x', k')$ of plaintext and key the joint distributions $D_{x,k}(R), D_{x',k'}(R)$ of d intermediate results induced by these pairs must be identical. Chari et al., on the other hand only demand that the distributions $D_{x,k}(R), D_{x',k'}(R)$ must be indistinguishable by an adversary. As Chari et al. point out, if the joint distributions of d intermediate results induced by different plaintext/key pairs are indistinguishable for an adversary then power analysis and timing attacks using information about at most d intermediate results cannot be mounted. Clearly, identical distributions are indistinguishable. Hence, an algorithm that is order-d perfectly masked is secure against timing and power analysis attacks using information about d intermediate results.

In the sequel, we will concentrate on methods to achieve a perfectly masked algorithm to compute AES. From the discussion above it follows that the perfectly masked algorithm for AES that we describe in Section 4.3 (page 33) is secure against timing and power analysis attacks using a single intermediate result. As can easily be seen, our algorithm is not secure, if an adversary has access to two or more intermediate results. Notice that most countermeasures proposed so far also assume an adversary with access to a single intermediate result, see (Akkar and Giraud 2001), (Golić and Tymen 2002) and (Trichina 2003).

4.2 Masking AES

(Messerges 2000) introduces the idea of masking all intermediate values of an encryption operation as an effective countermeasure against Simple Power Attacks and Differential Power Attacks. Randomizing the computation of a function f is, thus, achieved as $f(u')$ where $u' = u + r$ and r is a randomly chosen mask. If the function is linear, one can recover the desired value $f(u)$ from $f(u') = f(u) + f(r)$. A similar computation will recover $f(u)$ if the function f is affine. For non-linear functions, the previous equation does not hold true and it is necessary to come up with a series of computations depending only on r and u' such that we obtain the value of $f(u)$ without leaking any information.

We notice that in the case of the AES, the only non-linear function in the algorithm is the AES `SubBytes` transformation. As described in Section 2.3 (page 9), `SubBytes` consists of the function

$$\mathrm{INV}(x) = \begin{cases} x^{-1} & \text{, if } x \in \mathbb{F}_{256}^{\times} \\ 0 & \text{, if } x = 0 \end{cases}$$

together with an affine mapping. In particular, most researchers have concentrated their efforts on efficient methods to perform inversion over $\mathbb{F}_{256}$ in a secure manner via masking countermeasures, i.e., computing $u^{-1} + r$ from $u + r$ without compromising the value of u. In this context, three masking methods have been proposed: two of them, (Akkar and Giraud 2001) and (Golić and Tymen 2002) are based on the idea of combining boolean and multiplicative masking operations and the third one is based on the idea of masking the individual logic operations required to compute a $\mathbb{F}_{256}$ inverse. A simplification of (Akkar and Giraud 2001) was introduced in (Trichina et al. 2002) but it has been recently found by (Akkar, Bévan and Goubin 2004) that the simplifications lead to further vulnerabilities against DPA. Thus, we do not consider it any further. In the following, we shortly summarize the previously proposed countermeasures.

The Transform Masking Method (TMM)

In (Akkar and Giraud 2001), Akkar and Giraud introduce the Transform Masking Method (TMM) and algorithms to transform between boolean masking (XOR operation) and multiplicative masking (multiplication in $\mathbb{F}_{256}$) which is compatible with inversion in $\mathbb{F}_{256}$. (Akkar and Giraud 2001) solves the problem using Algorithm 6, where $r_1 \in \mathbb{F}_{256}$ is a random field element and $r_2 \in \mathbb{F}_{256}^{\times}$ is a random element of the multiplicative group.

However, as noticed in (Trichina et al. 2002) and (Golić and Tymen 2002), this countermeasure is susceptible to first-order DPA if $u = 0$ because zero cannot be masked with a multiplicative mask. It is clear that because of the special nature of the zero value, multiplicative masking cannot lead to perfect masking.

Algorithm 6 Transform Masking Method

Input: $u' = u \oplus r_1 \in \mathbb{F}_{256}$, $r_1 \in \mathbb{F}_{256}$, $r_2 \in \mathbb{F}_{256}^{\times}$
Output: $\mathrm{INV}(u) \oplus r_1$

1: $t_1 \leftarrow u' \cdot r_2$ $\{t_1 = (u \oplus r_1) \cdot r_2\}$
2: $t_2 \leftarrow r_1 \cdot r_2$ $\{t_2 = r_1 \cdot r_2\}$
3: $t_1 \leftarrow t_1 \oplus t_2$ $\{t_1 = u \cdot r_2\}$
4: $t_3 \leftarrow r_2^{-1}$ $\{t_3 = r_2^{-1}\}$
5: $t_1 \leftarrow \mathrm{INV}(t_1)$ $\{t_1 = \mathrm{INV}(u \cdot r_2)\}$
6: $t_2 \leftarrow t_3 \cdot r_1$ $\{t_2 = r_1 \cdot r_2^{-1}\}$
7: $t_1 \leftarrow t_1 \oplus t_2$ $\{t_1 = \mathrm{INV}(u \cdot r_2) \oplus (r_1 \cdot r_2^{-1})\}$
8: $t_1 \leftarrow t_1 \cdot r_2$ $\{t_1 = \mathrm{INV}(u) \oplus r_1\}$

Embedded Multiplicative Masking (EMM)

Let $m = x^8 + x^4 + x^3 + x + 1 \in \mathbb{F}_2[x]$ be the polynomial of the AES specification. The basic idea of EMM as described in (Golić and Tymen 2002) is to embed the field $\mathbb{F}_{256} = \mathbb{F}_2[x]/\langle m \rangle$ in the ring

$$\mathcal{R}_n := \mathbb{F}_2[x]/(m \cdot q) \cong \mathbb{F}_{256} \times \mathbb{F}_{2^n},$$

where $q \in \mathbb{F}_2[x]$ is another irreducible polynomial of degree n that is co-prime to m. The field $\mathbb{F}_{256}$ is a subring of the ring $\mathcal{R}_n$ with the homomorphism defined by

$$\begin{array}{rcl} \mathbb{F}_{256} & \rightarrow & \mathcal{R}_n \\ v & \mapsto & (v \bmod m, v \bmod q). \end{array}$$

(Golić and Tymen 2002), then, suggests to use a random mapping ρ_k defined by

$$\begin{array}{rcl} \rho_k : \mathbb{F}_{256} & \rightarrow & \mathcal{R}_n \\ v & \mapsto & v + rm \bmod mq \end{array}$$

where $r \in \mathbb{F}_2[x]$ is a randomly chosen polynomial of degree less than n. To compute $\mathrm{INV}(v)$ an adapted function

$$\begin{array}{rcl} \mathrm{INV}' : \mathbb{F}_{256} & \rightarrow & \mathbb{F}_{256} \\ v & \mapsto & v^{254} \bmod mq \end{array}$$

can be used.

In this way, arithmetic operations remain compatible with $\mathbb{F}_{256}$ and the zero value gets mapped to one of 2^n random values. Thus, it is harder to detect the zero value as n becomes larger. From a security point of view, however, the approach in (Golić and Tymen 2002) does not yield perfect masking since the sets of representatives of different values are pairwise disjoint. From an implementation point of view, we will show in Section 4.4.2 (page 39) that this method is too expensive to implement in hardware. This is important since our method can be implemented with less than half the hardware resources and, at the same time, yields perfect masking.

Combinational Logic Design for the AES Sbox on Masked Data

To the authors' knowledge, (Trichina 2003) is the first to consider embedding a masking countermeasure directly in hardware. (Trichina 2003) allows for a modified inversion function which on input $u \oplus r_1$ outputs $u^{-1} \oplus r_2$, where r_1 and r_2 need not be the same. In addition, (Trichina 2003) reduces the masking problem for inversion in $\mathbb{F}_{2^m}$ to the problem of masking a logical AND operation since masking XOR operations is, in principle, trivial. In particular, given masked bits $u' = u \oplus r_1$, $v' = v \oplus r_2$ and corresponding masks r_1, r_2, we compute $(u \wedge v) \oplus r_3$, where r_3 is the output mask. According to (Trichina 2003) and setting $r_3 = r_1 \wedge r_2$ this can be accomplished as:

$$(u \wedge v) \oplus r_3 = (u \wedge v) \oplus (r_1 \wedge r_2) = (u' \wedge v') \oplus \big((r_1 \wedge v') \oplus (r_2 \wedge u')\big) \tag{4.1}$$

where the parenthesis indicate the order in which intermediate results are computed. Equation (4.1) implies that we can compute the AND operation of two bits u, v without using the actual bits but rather their masked counterparts u', v' and corresponding masks r_1, r_2. We notice that if $u = v = 0$, the intermediate value $(r_1 \wedge v') \oplus (r_2 \wedge u')$ is always equal to zero for any value of r_1 and r_2. This implies that (4.1) does not lead to perfect masking.

4.3 Perfectly Masking AES against Order-1 Adversaries

As mentioned before, in order to obtain a perfectly masked algorithm for AES we concentrate on the problem of computing multiplicative inverses in $\mathbb{F}_{256}$ because this is the main step of the `SubBytes` transformation. In this section we present an algorithm that is secure against an adversary who is able to get the value of a single intermediate result. In Section 4.5 (page 41) we will show how to generalize this method to protect against order-d adversaries for an arbitrary but fixed $d \geq 1$.

Let r, r' be independent and uniformly distributed random masks. We start with an additively masked value $u \oplus r$ and would like to compute $\mathrm{INV}(u) \oplus r'$. However, a direct application of INV leads to $\mathrm{INV}(u \oplus r)$ that is of no use because of the non-linearity of inversion.

4.3.1 Idea

The basis of our idea is to compute $\mathrm{INV}(x)$ as x^{254} in $\mathbb{F}_{256}$. For simplicity we only consider the repeated squaring algorithm to compute the 254th power. However, to improve efficiency one could use an optimal addition chain. For a thorough treatment of efficient exponentiation methods see for example (von zur Gathen and Nöcker 1997, von zur Gathen and Gerhard 2003). In general the multiplicative inverse of an element over an arbitrary finite field $\mathbb{F}_{p^m}$ can always be computed by raising it to the (p^m-2)-th power. Since our inputs are additively masked values $(u \oplus r)$ we correct the result of every single operation in the repeated squaring algorithm in order to obtain the desired result. Our invariant is that at the end of each step our result has the form

$$(u^e \oplus r') \tag{4.2}$$

for some $e \in \mathbb{N}$ and $r' \in \mathbb{F}_{256}$ chosen uniformly at random. Hence, the problem is to correct the intermediate results without revealing any information about u.

4.3.2 Method

We introduce some variables: We name $r_{j,i}$ the jth random mask used in step i of the repeated squaring algorithm. All $r_{j,i}$ are independent and uniformly distributed masks. The direct result of a squaring or multiplication performed on some masked values is called f_i. Furthermore, we need so called auxiliary terms $s_{1,i}$ and $s_{2,i}$ to transform the direct result f_i. The variable $t_{1,i}$ is the intermediate result that appears during the correction and t_i is the final result which complies with our invariant (4.2), i.e., it is of the form $u^e \oplus r_{1,i}$ for some e.

The input to our modified inversion algorithm is the masked value $(u \oplus r_{1,0})$. Next, we describe how to perform multiplications and squarings in a perfectly masked manner. The security analysis is shown in Section 4.3.3. We distinguish between squaring and multiplication because the former is linear and hence can be masked more efficiently.

Perfectly Masked Squaring (PMS) The perfectly masked squaring algorithm that is used in step i of the repeated squaring algorithm is described in Algorithm 7. The input $t_{i-1} = u^e \oplus r_{1,i-1}$ is squared in step 1. In order to compute the output that respects our invariant we have to change the mask to $r_{1,i}$. To do so in steps 2 and 3 we use the auxiliary term $s_{1,i}$ and compute the desired output $t = u^{2e} \oplus r_{1,i}$.

Algorithm 7 Perfectly Masked Squaring (PMS)

Input: $t_{i-1} = u^e \oplus r_{1,i-1}$, $r_{1,i-1}$, $r_{1,i} \in \mathbb{F}_{256}$
Output: $u^{2e} \oplus r_{1,i} \in \mathbb{F}_{256}$
1: $f_i \leftarrow t_{i-1}^2$ $\{f_i = u^{2e} \oplus r_{1,i-1}^2\}$
2: $s_{1,i} \leftarrow r_{1,i-1}^2 \oplus r_{1,i}$ {auxiliary term to correct f_i}
3: $t_i \leftarrow f_i \oplus s_{1,i}$ $\{t_i = u^{2e} \oplus r_{1,i}\}$

Perfectly Masked Multiplication (PMM) Our perfectly masked multiplication method is described in Algorithm 8. The inputs are two intermediate results: the output x of the previous step and a freshly masked value x' derived by securely changing the masked value from $u \oplus r_1$ to $u \oplus r_2$. In Step 1 we calculate the product f_i of two intermediate results. The variable f_i contains the desired power of u as well as some disturbing terms. In Steps 2-5 we compute the auxiliary terms $s_{1,i}$ and $s_{2,i}$. In the end (Steps 6 and 7) we eliminate the disturbing parts of f_i and transform it according to our invariant. This is done by simply adding up the two auxiliary terms $s_{1,i}$, $s_{2,i}$ and f_i.

Algorithm 8 Perfectly Masked Multiplication (PMM)

Input: $x = u^e \oplus r_{1,i-1}$, $x' = u \oplus r_{2,i}$, $r_{1,i-1}$, $r_{1,i}$, $r_{2,i} \in \mathbb{F}_{256}$
Output: $u^{e+1} \oplus r_{1,i} \in \mathbb{F}_{256}$

1: $f_i \leftarrow x \cdot x'$ $\quad \{f_i = u^{e+1} \oplus u^e \cdot r_{2,i} \oplus u \cdot r_{1,i-1} \oplus r_{1,i-1} \cdot r_{2,i}\}$
2: $v_{1,i} \leftarrow x' \cdot r_{1,i-1}$ $\quad \{v_{1,i} = u \cdot r_{1,i-1} \oplus r_{1,i-1} \cdot r_{2,i}\}$
3: $v_{2,i} \leftarrow v_{1,i} \oplus r_{1,i}$ $\quad \{ v_{2,i} = u \cdot r_{1,i-1} \oplus r_{1,i-1} \cdot r_{2,i} \oplus r_{1,i}\}$
4: $s_{1,i} \leftarrow v_{2,i} \oplus r_{1,i-1} \cdot r_{2,i}$ $\quad \{s_{1,i} = u \cdot r_{1,i-1} \oplus r_{1,i}\}$
5: $s_{2,i} \leftarrow x \cdot r_{2,i}$ $\quad \{s_{2,i} = u^e \cdot r_{2,i} \oplus r_{1,i-1} \cdot r_{2,i}\}$
6: $t_{1,i} \leftarrow f_i \oplus s_{1,i}$ $\quad \{t_{1,i} = u^{e+1} \oplus u^e \cdot r_{2,i} \oplus r_{1,i-1} \cdot r_{2,i} \oplus r_{1,i}\}$
7: $t_i \leftarrow t_{1,i} \oplus s_{2,i}$ $\quad \{t_i = u^{e+1} \oplus r_{1,i}\}$

Table 4.1 lists all intermediate results that occur during the computation of x^{254}.

4.3.3 Security Analysis

As defined in our security model we have to look at all intermediate results. For Algorithm 7 and Algorithm 8 we only have to analyze the distributions of the following intermediate results: $f_i, s_{1,i}, s_{2,i}, t_i, t_{1,i}, v_{1,i}, v_{2,i}$ where $1 \leq i \leq 13$. These are the results that depend on u. We can neglect intermediate results such as $r_{1,i}^2$ since they do not depend on u.

Our security analysis is based on the following three lemmata that characterize the distributions of intermediate results.

i	Op	f_i	$s_{1,i}$	$s_{2,i}$	$t_{1,i}$	t_i
1	(S)	$u^2 \oplus r_{1,0}^2$	$r_{1,0}^2 \oplus r_{1,1}$			$u^2 \oplus r_{1,1}$
2	(M)	$(u^2 \oplus r_{1,1})(u \oplus r_{2,2})$	$ur_{1,1} \oplus r_{1,2}$	$u^2 r_{2,2} \oplus r_{1,1}r_{2,2}$	$u^3 \oplus u^2 r_{2,2} \oplus r_{1,1}r_{2,2} \oplus r_{1,2}$	$u^3 \oplus r_{1,2}$
3	(S)	$u^6 \oplus r_{1,2}^2$	$r_{1,2}^2 \oplus r_{1,3}$			$u^6 \oplus r_{1,3}$
4	(M)	$(u^6 \oplus r_{1,3})(u \oplus r_{2,4})$	$ur_{1,3} \oplus r_{1,4}$	$u^6 r_{2,4} \oplus r_{1,3}r_{2,4}$	$u^7 \oplus u^6 r_{2,4} \oplus r_{1,3}r_{2,4} \oplus r_{1,4}$	$u^7 \oplus r_{1,4}$
5	(S)	$u^{14} \oplus r_{1,4}^2$	$r_{1,4}^2 \oplus r_{1,5}$			$u^{14} \oplus r_{1,5}$
6	(M)	$(u^{14} \oplus r_{1,5})(u \oplus r_{2,6})$	$ur_{1,5} \oplus r_{1,6}$	$u^{14} r_{2,6} \oplus r_{1,5}r_{2,6}$	$u^{15} \oplus u^{14} r_{2,6} \oplus r_{1,5}r_{2,6} \oplus r_{1,6}$	$u^{15} \oplus r_{1,6}$
7	(S)	$u^{30} \oplus r_{1,6}^2$	$r_{1,6}^2 \oplus r_{1,7}$			$u^{30} \oplus r_{1,7}$
8	(M)	$(u^{30} \oplus r_{1,7})(u \oplus r_{2,8})$	$ur_{1,7} \oplus r_{1,8}$	$u^{30} r_{2,8} \oplus r_{1,7}r_{2,8}$	$u^{31} \oplus u^{30} r_{2,8} \oplus r_{1,7}r_{2,8} \oplus r_{1,8}$	$u^{31} \oplus r_{1,8}$
9	(S)	$u^{62} \oplus r_{1,8}^2$	$r_{1,8}^2 \oplus r_{1,9}$			$u^{62} \oplus r_{1,9}$
10	(M)	$(u^{62} \oplus r_{1,9}^2)(u \oplus r_{2,10})$	$ur_{1,9} \oplus r_{1,10}$	$u^{62} r_{2,10} \oplus r_{1,9}r_{2,10}$	$u^{63} \oplus u^{62} r_{2,10} \oplus r_{1,9}r_{2,10} \oplus r_{1,10}$	$u^{63} \oplus r_{1,10}$
11	(S)	$u^{126} \oplus r_{1,10}^2$	$r_{1,10}^2 \oplus r_{1,11}$			$u^{126} \oplus r_{1,11}$
12	(M)	$(u^{126} \oplus r_{1,11})(u \oplus r_{2,12})$	$ur_{1,11} \oplus r_{1,12}$	$u^{126} r_{2,12} \oplus r_{1,11}r_{2,12}$	$u^{127} \oplus u^{126} r_{2,12} \oplus r_{1,11}r_{2,12} \oplus r_{1,12}$	$u^{127} \oplus r_{1,12}$
13	(S)	$u^{254} \oplus r_{1,12}^2$	$r_{1,12}^2 \oplus r_{1,13}$			$u^{254} \oplus r_{1,13}$

Table 4.1: Computation of $(u^{254} \oplus r_{1,13})$ using repeated squaring

Lemma 1 *Let $u \in \mathbb{F}_{256}$ be arbitrary. Let $r \in \mathbb{F}_{256}$ be uniformly distributed and independent of u. Then $Z = u \oplus r$ is uniformly distributed.*

Lemma 2 *Let $u, u' \in \mathbb{F}_{256}$ and $r, r' \in \mathbb{F}_{256}$ be independent and uniformly distributed. Set $I_1 = u \oplus r$ and $I_2 = u' \oplus r'$. Then the product $Z = I_1 \cdot I_2$ is distributed according to*

$$Pr(Z = b) = \begin{cases} (2^9 - 1)/2^{16} & , \textit{if } b = 0 \\ (2^8 - 1)/2^{16} & , \textit{if } b \neq 0 \end{cases}$$

We call this distribution D_0.

Lemma 3 *In any finite field of characteristic 2, squaring is a one-to-one mapping.*

The proofs of these lemmata are straightforward.

In the sequel, we examine each of the intermediate results that occur in the PMS (Algorithm 7) and in the PMM (Algorithm 8). We show that the distributions of each of these intermediate results is independent of the secret value u.

Analysis of f_i We have to look at the intermediate result f_i in the two cases of squaring and multiplication.

Squaring: The computation is $f_i \leftarrow t_{i-1}^2 = u^{2e} \oplus r_{1,i-1}^2$ for some $2 \leq e \leq 254$. Since $r_{1,i-1}$ is chosen uniformly at random, Lemma 3 together with Lemma 1 shows that f_i is uniformly distributed for all u.

Multiplication: The variable is computed as $f_i \leftarrow (u^e + r_{1,i-1}) \cdot (u \oplus r_{2,i}) = u^{e+1} \oplus u^e r_{2,i} \oplus u r_{1,i-1} \oplus r_{1,i-1} r_{2,i}$. Here the terms $u^e + r_{1,i-1}$ and $u \oplus r_{2,i}$ are independent (because of the independence of $r_{1,i-1}$ and $r_{2,i}$) and uniformly distributed (see Lemma 1). So by Lemma 2, f_i is distributed according to D_0 for all u.

Analysis of $s_{1,i}, s_{2,i}$ We examine the intermediate results $s_{1,i}, s_{2,i}$ for multiplication and squaring.

Squaring: Here $s_{1,i}$ can be neglected since it does not depend on u.

Multiplication: The variable $s_{1,i}$ is calculated by adding or multiplying independent masks on the term $(u \oplus r_{2,i})$ leading to the term $u r_{1,i-1} \oplus r_{1,i}$. So $s_{1,i}$ is obviously uniformly distributed. The variable $s_{2,i} \leftarrow (u^e \oplus r_{1,i-1}) \cdot r_{2,i}$ is the product of two independent and uniformly distributed variables that are both independent of u. So the variable $s_{2,i}$ is distributed according to D_0 independent of the value of u.

Analysis of $t_{1,i}, t_i$ All these intermediate results are sums of some part depending on u and an independent additive mask. So all of them are uniformly distributed by Lemma 1.

Hence corresponding intermediate results are always identically distributed and independent of the value of u. This implies that the whole computation is perfectly masked.

4.3.4 Simplified Version

Previously we assumed that for each step we generate new random masks. In the sequel, we show how to improve the method described above in terms of the number of random masks needed to achieve a perfectly masked exponentiation. The fact that an adversary only obtains a single intermediate result allows us to reuse random masks in different steps of the algorithm.

Algorithm 9 Simplified Perfectly Masked Squaring (s-PMS)

Input: $x = u^e \oplus r_1, r_1 \in \mathbb{F}_{256}$
Output: $u^{2e} \oplus r_1 \in \mathbb{F}_{256}$
1: $f_i \leftarrow x^2$ $\{f_1 = u^{2e} \oplus r_1^2\}$
2: $s_{1,i} \leftarrow r_1^2 \oplus r_1$ {auxiliary term to correct f_i}
3: $t_i \leftarrow f_i \oplus s_{1,i}$ $\{t_i = u^{2e} \oplus r_1\}$

We call the improved version of the squaring and multiplication algorithm *simplified Perfectly Masked Squaring (s-PMS)* (Algorithm 9) and *simplified Perfectly Masked Multiplication (s-PMM)* (Algorithm 10), respectively.

Algorithm 10 Simplified Perfectly Masked Multiplication (s-PMM)

Input: $x = u^e \oplus r_1$, $x' = u \oplus r_2$, $r_1, r_2, r_3 \in \mathbb{F}_{256}$
Output: $u^{e+1} \oplus r_1 \in \mathbb{F}_{256}$
1: $f_i \leftarrow x \cdot x'$ $\{f_i = u^{e+1} \oplus u^e \cdot r_2 \oplus u \cdot r_1 \oplus r_1 \cdot r_2\}$
2: $t_1 \leftarrow r_1 \cdot r_2 \oplus r_3$ $\{t_1 = r_1 \cdot r_2 \oplus r_3\}$
3: $f' \leftarrow f \oplus t_1$ $\{f' = u^{e+1} \oplus u^e \cdot r_2 \oplus u \cdot r_1 \oplus r_3\}$
4: $s_{1,i} \leftarrow x \cdot r_2$ $\{s_{1,i} = u^e \cdot r_1 \oplus r_1 \cdot r_2\}$
5: $s_{2,i} \leftarrow x' \cdot r_1$ $\{s_{2,i} = u \cdot r_1 \oplus r_1 \cdot r_2\}$
6: $t_{1,i} \leftarrow f_i' \oplus s_{1,i}$ $\{t_{1,i} = u^{e+1} \oplus u \cdot r_1 \oplus r_1 \cdot r_2 \oplus r_3\}$
7: $t_{2,i} \leftarrow t_{1,i} \oplus s_{2,i}$ $\{t_{2,i} = u^{e+1} \oplus r_3\}$
8: $t_{3,i} \leftarrow t_{2,i} \oplus r_3 \oplus r_1$ $\{t_{3,i} = u^{e+1} \oplus r_1\}$

Thus, we can reduce the number of random masks needed to only three masks (r_1, r_2, r_3). To achieve this we modify our computations such that after each step we switch back to our original mask. This can be done by simply adding our original mask and then adding our temporarily used mask. Because of the independence of the masks this has no impact on the security. Table 4.2 lists all intermediate results that occur during the computation of x^{254} using the simplified method.

i	Op	f_i	$s_{1,i}$	$s_{2,i}$	$t_{1,i}$	$t_{2,i}$	$t_{3,i}$	t_i
1	(S)	$u^2 \oplus r_1^2$	$r_1^2 \oplus r_1$					$u^2 \oplus r_1$
2	(M)	$(u^2 \oplus r_1)(u \oplus r_2)$	$ur_1 \oplus r_3$	$u^2 r_2 \oplus r_1 r_2$	$u^3 \oplus u^2 r_2 \oplus r_1 r_2 \oplus r_3$	$u^3 \oplus r_3$	$u^3 \oplus r_3 \oplus r_1$	$u^3 \oplus r_1$
3	(S)	$u^6 \oplus r_1^2$	$r_1^2 \oplus r_1$					$u^6 \oplus r_1$
4	(M)	$(u^6 \oplus r_1)(u \oplus r_2)$	$ur_1 \oplus r_3$	$u^6 r_2 \oplus r_1 r_2$	$u^7 \oplus u^6 r_2 \oplus r_1 r_2 \oplus r_3$	$u^7 \oplus r_3$	$u^7 \oplus r_3 \oplus r_1$	$u^7 \oplus r_1$
5	(S)	$u^{14} \oplus r_1^2$	$r_1^2 \oplus r_1$					$u^{14} \oplus r_1$
6	(M)	$(u^{14} \oplus r_1)(u \oplus r_2)$	$ur_1 \oplus r_3$	$u^{14} r_2 \oplus r_1 r_2$	$u^{15} \oplus u^{14} r_2 \oplus r_1 r_2 \oplus r_3$	$u^{15} \oplus r_3$	$u^{15} \oplus r_3 \oplus r_1$	$u^{15} \oplus r_1$
7	(S)	$u^{30} \oplus r_1^2$	$r_1^2 \oplus r_1$					$u^{30} \oplus r_1$
8	(M)	$(u^{30} \oplus r_1)(u \oplus r_2)$	$ur_1 \oplus r_3$	$u^{30} r_2 \oplus r_1 r_2$	$u^{31} \oplus u^{30} r_2 \oplus r_1 r_2 \oplus r_3$	$u^{31} \oplus r_3$	$u^{31} \oplus r_3 \oplus r_1$	$u^{31} \oplus r_1$
9	(S)	$u^{62} \oplus r_1^2$	$r_1^2 \oplus r_1$					$u^{62} \oplus r_1$
10	(M)	$(u^{62} \oplus r_1^2)(u \oplus r_2)$	$ur_1 \oplus r_3$	$u^{62} r_2 \oplus r_1 r_2$	$u^{63} \oplus u^{62} r_2 \oplus r_1 r_2 \oplus r_3$	$u^{63} \oplus r_3$	$u^{63} \oplus r_3 \oplus r_1$	$u^{63} \oplus r_1$
11	(S)	$u^{126} \oplus r_1^2$	$r_1^2 \oplus r_1$					$u^{126} \oplus r_1$
12	(M)	$(u^{126} \oplus r_1)(u \oplus r_2)$	$ur_1 \oplus r_3$	$u^{126} r_2 \oplus r_1 r_2$	$u^{127} \oplus u^{126} r_2 \oplus r_1 r_2 \oplus r_3$	$u^{127} \oplus r_3$	$u^{127} \oplus r_3 \oplus r_1$	$u^{127} \oplus r_1$
13	(S)	$u^{254} \oplus r_1^2$	$r_1^2 \oplus r_1$					$u^{254} \oplus r_1$

Table 4.2: Computation of $(u^{254} \oplus r_1)$ using repeated squaring (simplified version)

4.4 Implementation and Costs

Throughout the chapter, we have only considered a theoretical implementation of the inversion algorithm according to the square-and-multiply algorithm. However, our method is compatible with any implementation that combines additions, multiplications, and squarings in a field or ring. More precisely, an arbitrary straight-line program over some finite field using only additions and multiplications can be transformed to an equivalent program that is perfectly masked. We do not consider software implementations of the presented countermeasures. However, we notice that for constrained environments previous publications have based their software implementations of side channel countermeasures on table lookups. From a hardware point of view, the most area efficient ASIC hardware implementation is the one described in (Satoh, Morioka, Takano and Munetoh 2001) based on composite fields. We will discuss a possible implementation of our countermeasure based on composite fields and will provide area and delay estimates in the next section.

4.4.1 Efficient Hardware Implementation over $GF(((2^2)^2)^2)$

First we describe in some detail how to implement an inverter over $GF(((2^2)^2)^2)$, so that it is clear how we obtained our area and delay estimates. This methodology is not new and it is well known in the literature, e.g., see (Lidl and Niederreiter 1983). We assume a composite field representation $GF(((2^2)^2)^2) \cong \mathbb{F}_{256}$ for the inverse transformation using the following irreducible polynomials:

$$\begin{array}{ll} GF(2^2) & : P(x) = x^2 + x + 1,\ \ P(\alpha) = 0 \\ GF((2^2)^2) & : Q(y) = y^2 + y + \alpha,\ \ Q(\beta) = 0 \\ GF(((2^2)^2)^2) & : R(z) = z^2 + z + \lambda,\ \ \lambda = (\alpha + 1)\beta \end{array}$$

We use the s-PMM and s-PMS algorithms from Section 4.3 instead of the usual ones to build our inversion circuit and, thus, render it secure against side channel attacks. Based on (Itoh and Tsujii 1988) and (Guajardo and Paar 2002), (Satoh et al. 2001) notice that for $A \in GF(((2^2)^2)^2)$, A^{-1} can be computed as $A^{-1} = (A^{17})^{-1}A^{16}$, where $A^{17} \in GF((2^2)^2)$. See for example (Lidl and Niederreiter 1983) for the proof. Notice that the Itoh and Tsujii algorithm can be recursively applied to $B = A^{17} \in GF((2^2)^2)$, thus obtaining $B^{-1} = (B^4 \cdot B)^{-1} \cdot (B^4)$ where $B^5 \in GF(2^2)$. In the following, we write $B = B_1\beta{+}B_0 \in GF((2^2)^2)$ with $B_i \in GF(2^2)$. Then, we can minimize the area requirement of the implementation using the following facts:

1. $B^4 \in GF((2^2)^2)$ can be computed as $B^4 \equiv B_1\beta + (B_1 + B_0)$, i.e., only one addition over $GF(2^2)$.
2. $B^5 \in GF(2^2)$ can be computed as $B^5 \equiv B_0 \cdot B_1 + B_0^2 + B_1^2 \cdot \alpha$, where $B_1^2 \cdot \alpha$ requires only wires for its implementation (no gates).
3. Given $C = c_1\alpha + c_0 \in GF(2^2)$, $C^{-1} \equiv c_1\alpha + (c_1 + c_0)$, i.e., it requires one $GF(2)$ adder.

Thus, computing $B^{-1} = B^{-5} \cdot B^4 \in GF((2^2)^2)$ requires 3 $GF(2^2)$ multipliers, 1 $GF(2^2)$ squarer, and 4 $GF(2^2)$ adders. The inversion in $GF(((2^2)^2)^2)$ can then be implemented according to (Satoh et al. 2001) with 2 adders, 3 multipliers, 1 inverter, and 1 squarer followed by multiplication with $\lambda = (\alpha + 1)\beta$, all over $GF((2^2)^2)$.

The hardware implementation of the perfectly masked version can be implemented similarly except that instead of using the usual adders, multipliers, squarers, and inverters, we use circuits which implement the algorithms from Section 4.3 (page 33).

4.4.2 Cost and Comparison to Previous Countermeasures

Area and delay estimates for circuits with and without countermeasures are provided in Table 4.3. The estimates are given in terms of the area and delay of 2-input AND gates, 2-input XOR gates, and NOT gates. The complexity and specific implementation of these circuits is taken from (Voigtländer 2003). In addition, we provide complexity estimates in terms of normalized area and delay. The normalization is done with respect to the area and delay of a NOT gate. We have assumed that the areas of a 2-input AND gate and 2-input XOR gate are twice and 3 times that of an inverter, respectively. Similarly, it is assumed that the delays of NOT, AND, and XOR gates are equal. Notice that the assumptions regarding the gates' area and delay are not arbitrary but based on the actual sizes of several standard cell libraries.

Arithmetic Operation	A	A'	T	T'	$A' \cdot T'$
Inversion over $GF(((2^2)^2)^2)$ (Satoh et al. 2001)	312	1	17	1	1
Inversion with DPA countermeasure from (Trichina 2003) according to (4.1)	1071	3.4	26	1.5	5.1
$GF(((2^2)^2)^2)$ PM inverter from this thesis (Blömer et al. 2004)	1704	5.5	21	1.2	6.6
Inversion with DPA countermeasure from (Trichina 2003)	1341	4.3	34	2	8.6
Inversion with countermeasure from (Akkar and Giraud 2001)	1784	5.7	34	2	11.4

Table 4.3: Hardware cost comparison of area A and delay T for different inversion circuits with side channel countermeasures. $A' := A/A_{Normal\ Inv.}$ and $T' := T/T_{Normal\ Inv.}$ are the normalized area and delay respectively.

Finally, we point out that (Satoh et al. 2001) which describes AES ASIC implementations over $GF(((2^2)^2)^2)$ does not provide the actual circuits used to implement the AES sbox.

Table 4.3 provides a cost comparison among the different masking countermeasures. We did not consider the method from (Golić and Tymen 2002) briefly sketched in Section 4.2 (page 32) because its hardware implementation requires too many hardware resources. We can estimate the cost of (Golić and Tymen 2002) if the degree of the polynomial q is $n = 8$ by simply considering the cost of a multiplier and an inverter over $\mathbb{F}_2[x]/(mq) \cong \mathbb{F}_{256} \times \mathbb{F}_{2^n}$. According to (Drolet 1998), such a multiplier requires 289 2-input AND gates and 272 2-input XOR gates. The map $\mathrm{INV}'(v) = v^{254} \bmod mq$ can also be implemented with a multiplier (a squarer requires only wires). Thus, we would need at least 1 multiplier and 1 inverter over $\mathbb{F}_2[x]/(mq)$ and 3 multipliers and 1 inverter over $\mathbb{F}_{256}$. This results in a circuit which requires at least 731 AND and 766 XOR gates or about twice as many gates as our method.

Table 4.3 shows that the countermeasure of (Trichina 2003) implemented according to Equation (4.1) on page 33 has the best area/time product of all the implementations. However, as we have seen in Section 4.2, this countermeasure is susceptible to DPA attacks if the input byte is zero and, thus, does not provide perfect masking. If we then consider the best area/time product of the countermeasures that offer DPA resistance, the implementation presented in this chapter has the best area/time product. This result is mainly due to the reduced critical path in the circuit. In addition, our design encourages re-usability of previously designed blocks. In other words, since the masking method depends only on multipliers and adders, if one has multiplier and adder blocks already designed, they can be used immediately to build a perfectly masked circuit (with the work from (Trichina 2003), implementation of the masking countermeasure would require a complete circuit redesign).

Finally, we estimate the cost that our masking countermeasure would have on an AES hardware implementation. To do this, we assume that the implementation would follow the architecture described in (Satoh et al. 2001) where the `SubBytes` transformation occupies about 22% of the design with 4 sboxes in parallel. In `SubBytes`, the inverse transformation accounts for 60% or about 14% of the total area. We also assume that the remaining circuits require twice as much area as an implementation without masking countermeasures. Then, our new inversion circuit would need about 2.5 times the area that an AES hardware implementation without countermeasures would need. Of this 31% would correspond to the inverter circuit. The required area is only 20% larger than an implementation that uses hardware countermeasures based on the usage of different hardware logic. Such methods double the hardware resources when compared to an implementation using standard (single-rail) logic.

In addition to time and area, other costs are also of importance. For example, the amount of randomness is rather crucial since its generation is quite expensive. In our simplified algorithm we only need 3 random masks in order to compute INV(x) in a secure manner. Another important cost factor is the number of operations that have to be protected by hardware means. Our approach needs this inevitable protection only for one intermediate result. Hence it is optimal with respect to this cost measure.

4.5 Order-d Perfectly Masking

For the sake of completeness, in this section we focus on generalizing the method of Section 4.3 to adversaries of arbitrary but fixed order d. However, adversaries that can obtain values of two or more intermediate results are very powerful and assumed to be not realistic right now. Moreover, for increasing d an increasing amount of random bits is needed to achieve such a high level of security. This, however, decreases efficiency considerably. In particular, instead of using a single random byte r as a mask one has to use masks of the form

$$R = \sum_{i=1}^{d} r_i$$

that are the sum of d independent and uniformly distributed random bytes. Hence, our invariant in the order-d case is that every output of an operation is of the form

$$u \oplus R = u \oplus \sum_{i=1}^{d} r_i \tag{4.3}$$

for d independent and uniformly distributed random bytes r_i.

4.5.1 Perfect Mask Change

However, simply substituting the mask r by mask R in the method described above is not sufficient. To see this, consider the problem of changing masks of intermediate results in order to introduce new randomness into the encryption. Note that changing masks is implicitely done in the Perfectly Masked Squaring Algorithm (Algorithm 7 (page 34)) and the Perfectly Masked Multiplication Algorithm (Algorithm 8 (page 35)). To change the mask R of an intermediate result $Z_1 := u \oplus R$ into the mask R' the straightforward approach is to compute

1. $Z_1 := u \oplus R^{(1)}$
2. $Z_2 := Z_1 \oplus R^{(2)} = u \oplus R^{(1)} \oplus R^{(2)}$
3. $Z_3 := Z_2 \oplus R^{(1)} = u \oplus R^{(2)}$

However, for $d \geq 3$ an order-d adversary $\mathcal{A}$ can get the values of Z_1, Z_2 and Z_3. Hence, $\mathcal{A}$ can compute the unmasked value $u = Z_1 \oplus Z_2 \oplus Z_3$.

In order to securely change the mask $R^{(1)} = \sum_{i=1}^{d} r_i^{(1)}$ of an intermediate result $u \oplus R^{(1)}$ to a different mask $R^{(2)} = \sum_{i=1}^{d} r_i^{(2)}$ we propose to use Algorithm 11.

Algorithm 11 Perfect Mask Change (PMC)

Input: $Z_1 = u^e \oplus R^{(1)}$ for some $1 \leq e \leq 254$, $d \in \mathbb{N}, \underbrace{r_1 \ldots, r_d}_{R^{(1)}}, \underbrace{r_{d+1}, \ldots, r_{2d}}_{R^{(2)}}$

Output: $Z_{2d+1} = u^e \oplus R^{(2)}$

1: **for** $i = 2 \ldots 2d$ **do**
2: $\quad Z_i \leftarrow Z_{i-1} \oplus r_{d+i}$ {add i-th masking byte of $R^{(2)}$}
3: $\quad Z_{i+1} \leftarrow Z_i \oplus r_i$ {remove i-th masking byte of $R^{(1)}$}
4: $\quad i \leftarrow i + 1$
5: **end for**

Example 1 *For $d = 3$ Algorithm 11 computes the following intermediate results:*

$$\begin{array}{lllllllllllll}
Z_1 & = & u^e & \oplus & r_1^{(1)} & \oplus & r_2^{(1)} & \oplus & r_3^{(1)} & & & & \\
Z_2 & = & u^e & \oplus & r_1^{(1)} & \oplus & r_2^{(1)} & \oplus & r_3^{(1)} & \oplus & r_1^{(2)} & & \\
Z_3 & = & u^e & & & \oplus & r_2^{(1)} & \oplus & r_3^{(1)} & \oplus & r_1^{(2)} & & \\
Z_4 & = & u^e & & & \oplus & r_2^{(1)} & \oplus & r_3^{(1)} & \oplus & r_1^{(2)} & \oplus & r_2^{(2)} \\
Z_5 & = & u^e & & & & & \oplus & r_3^{(1)} & \oplus & r_1^{(2)} & \oplus & r_2^{(2)} \\
Z_6 & = & u^e & & & & & \oplus & r_3^{(1)} & \oplus & r_1^{(2)} & \oplus & r_2^{(2)} \quad \oplus \quad r_3^{(2)} \\
Z_7 & = & u^e & & & & & & & \oplus & r_1^{(2)} & \oplus & r_2^{(2)} \quad \oplus \quad r_3^{(2)}
\end{array}$$

Security Analysis

We first introduce our notation that we use for the proof of security. Let d be the number of intermediate results an adversary can get. For $1 \leq i \leq 2d + 1$ let

$$\delta_i = i + 1 \mod 2$$

indicate whether the intermediate result Z_i is randomized by d or $d+1$ masks. Let

$$S_i = \left\{ \left\lfloor \frac{i}{2} \right\rfloor, \ldots, \left\lfloor \frac{i}{2} \right\rfloor + d + \delta_i \right\}$$

denote the set of indices of masks involved in the randomization of Z_i. I.e.,

$$Z_i = u^e \oplus \sum_{j \in S_i} r_j.$$

Furthermore, let $1 \leq \ell \leq d$ and

$$I := \{i_1, \ldots, i_\ell \mid i_1 < i_2 < \cdots < i_\ell\}$$

be the set of indices of intermediate results known to the attacker and let

$$M := \{j_1, \ldots, j_{d-\ell} \mid j_1 < j_2 < \cdots < j_{d-\ell}\}$$

be the set of indices of masking bytes known to the attacker.

For $i \in I$ let $\overline{S}_i = S_i \setminus M$ denote the set of masks unknown to the attacker that randomize the intermediate result Z_i and let

$$\overline{Z}_i := Z_i \oplus \sum_{j \in M \cap S_i} r_j = u^e \oplus \sum_{j \in \overline{S}_i} r_j$$

denote a known intermediate result after removing all known masks. Note that $|\overline{S}_i| \geq 1$ for all $i \in I$ holds by construction. Hence, all $\overline{Z}_i$ are uniformly distributed by Lemma 1 (page 36). Furthermore, depending on the set of known masks it is possible that $\overline{Z}_i = \overline{Z}_j$ for $Z_i \neq Z_j$.

Lemma 4 *Let Z_i, $\overline{Z}_i$ and r_j be defined as above. Then*

$$\Pr(Z_{i_1}, \ldots, Z_{i_\ell} \mid r_{j_1}, \ldots, r_{j_{d-\ell}}) = \Pr(\overline{Z}_{i_1}, \ldots, \overline{Z}_{i_\ell}).$$

Proof. Let $\zeta_{i_1}, \ldots, \zeta_{i_\ell} \in \mathbb{F}_{256}$ and $\rho_{j_1}, \ldots, \rho_{j_{d-\ell}} \in \mathbb{F}_{256}$.

$$\begin{aligned}
&\Pr\left(Z_{i_1} = \zeta_{i_1}, \ldots, Z_{i_\ell} = \zeta_{i_\ell} \mid r_{j_1} = \rho_{j_1}, \ldots, r_{j_{d-\ell}} = \rho_{j_{d-\ell}}\right) \\
= \; &\Pr\left(\left(\overline{Z}_{i_1} \oplus \sum_{j \in S_{i_1} \cap M} r_j = \zeta_{i_1} \right), \ldots, \left(\overline{Z}_{i_\ell} \oplus \sum_{j \in S_{i_\ell} \cap M} r_j = \zeta_{i_\ell} \right) \;\middle|\; (r_{j_1} = \rho_{j_1}), \ldots, (r_{j_{d-\ell}} = \rho_{j_{d-\ell}}) \right) \\
= \; &\Pr\left(\left(\overline{Z}_{i_1} = \zeta_{i_1} \oplus \sum_{j \in S_{i_1} \cap M} \rho_j \right), \ldots, \left(\overline{Z}_{i_\ell} = \zeta_{i_\ell} \oplus \sum_{j \in S_{i_\ell} \cap M} \rho_j \right) \;\middle|\; (r_{j_1} = \rho_{j_1}), \ldots, (r_{j_{d-\ell}} = \rho_{j_{d-\ell}}) \right) \\
= \; &\frac{\Pr\left(\left(\overline{Z}_{i_1} = \zeta_{i_1} \oplus \sum_{j \in S_{i_1} \cap M} \rho_j \right), \ldots, \left(\overline{Z}_{i_\ell} = \zeta_{i_\ell} \oplus \sum_{j \in S_{i_\ell} \cap M} \rho_j \right), (r_{j_1} = \rho_{j_1}), \ldots, (r_{j_{d-\ell}} = \rho_{j_{d-\ell}}) \right)}{\Pr\left((r_{j_1} = \rho_{j_1}), \ldots, (r_{j_{d-\ell}} = \rho_{j_{d-\ell}}) \right)}
\end{aligned}$$

Since $\mid \overline{S}_i \mid \geq 1$ for all $i \in I$ the variables $\overline{Z}_{i_1}, \ldots, \overline{Z}_{i_\ell}$ and $r_j, \ldots, r_{j_{d-\ell}}$ are stochastically independent. Hence, we have that

$$\frac{\Pr\left(\left(\overline{Z}_{i_1} = \zeta_{i_1} \oplus \sum_{j \in S_{i_1} \cap M} \rho_j\right), \ldots, \left(\overline{Z}_{i_\ell} = \zeta_{i_\ell} \oplus \sum_{j \in S_{i_\ell} \cap M} \rho_j\right), (r_{j_1} = \rho_{j_1}), \ldots, (r_{j_{d-\ell}} = \rho_{j_{d-\ell}})\right)}{\Pr\left((r_{j_1} = \rho_{j_1}), \ldots, (r_{j_{d-\ell}} = \rho_{j_{d-\ell}})\right)}$$

$$= \frac{\Pr\left(\left(\overline{Z}_{i_1} = \zeta_{i_1} \oplus \sum_{j \in S_{i_1} \cap M} \rho_j\right), \ldots, \left(\overline{Z}_{i_\ell} = \zeta_{i_\ell} \oplus \sum_{j \in S_{i_\ell} \cap M} \rho_j\right)\right) \cdot \Pr\left((r_{j_1} = \rho_{j_1}), \ldots, (r_{j_{d-\ell}} = \rho_{j_{d-\ell}})\right)}{\Pr\left((r_{j_1} = \rho_{j_1}), \ldots, (r_{j_{d-\ell}} = \rho_{j_{d-\ell}})\right)}$$

$$= \Pr\left(\left(\overline{Z}_{i_1} = \zeta_{i_1} \oplus \sum_{j \in S_{i_1} \cap M} \rho_j\right), \ldots, \left(\overline{Z}_{i_\ell} = \zeta_{i_\ell} \oplus \sum_{j \in S_{i_\ell} \cap M} \rho_j\right)\right)$$

Since all $\overline{Z}_i$ for $1 \leq i \leq \ell$ are uniformly distributed, this proves the lemma. □

To prove the security of Algorithm 11 we also need the following lemma.

Lemma 5 *For some $1 \leq b \leq \ell$ let*

$$I = \bigcup_{1 \leq c \leq b} I_c$$

be a partition of the set I into subsets $I_1, \ldots, I_b$ such that

$$\overline{Z}_i = \overline{Z}_j \;\Leftrightarrow\; \exists\, 1 \leq c \leq b \;:\; i, j \in I_c.$$

I.e., the indices i, j of two elements Z_i are in the same subset I_c iff $\overline{Z_i} = \overline{Z_j}$.

Then

$$\Pr\left(\bigwedge_{i \in I} \overline{Z}_i\right) = \prod_{1 \leq c \leq b} \Pr\left(\bigwedge_{i \in I_c} \overline{Z}_i\right).$$

Proof. For $i \in I_c$ let $\overline{T}_c$ denote the set of indices of masks that randomize $\overline{Z}_i$. The construction of the intermediate results of Algorithm 11 implies that for each $1 \leq c < b$

$$\min\{j \mid j \in \overline{T}_c\} < \min\{j \mid j \in \bigcup_{c < c' \leq b} \overline{T}_{c'}\}$$

or

$$\max\{j \mid j \in \overline{T}_c\} < \max\{j \mid j \in \bigcap_{c < c' \leq b} \overline{T}_{c'}\}$$

holds. Hence, for each $1 \leq c < b$ at least one of the following cases holds:

1. If

$$\overline{T}_c \setminus \bigcup_{c+1 \leq j \leq b} \overline{T}_j \neq \emptyset$$

it follows that all elements of $\{\overline{Z}_i | i \in I_c\}$ are randomized by a uniformly distributed mask that is not involved in randomizing elements of $\{\overline{Z}_i | i \in \bigcup_{c+1 \leq j \leq b} I_j\}$. Hence, it follows that

$$\Pr\left(\bigwedge_{i \in \bigcup_{c \leq j \leq b} I_j} \overline{Z}_i\right) = \Pr\left(\bigwedge_{i \in I_c} \overline{Z}_i\right) \cdot \Pr\left(\bigwedge_{i \in \bigcup_{c+1 \leq j \leq b} I_j} \overline{Z}_i\right).$$

2. If
$$\bigcap_{c+1 \le j \le b} \overline{T}_j \setminus \overline{T}_c \neq \emptyset$$
it follows that all elements of $\{\overline{Z}_i | i \in \bigcup_{c+1 \le j \le b} I_j\}$ are randomized by a uniformly distributed mask that is not involved in randomizing elements of $\{\overline{Z}_i | i \in I_c\}$. Hence, it follows that
$$\Pr\left(\bigwedge_{i \in \bigcup_{c \le j \le b} I_j} \overline{Z}_i\right) = \Pr\left(\bigwedge_{i \in I_c} \overline{Z}_i\right) \cdot \Pr\left(\bigwedge_{i \in \bigcup_{c+1 \le j \le b} I_j} \overline{Z}_i\right).$$

Applying this case differentiation inductively proves the lemma. □

Lemma 4 shows that instead of analyzing the joint distribution of ℓ intermediate results $Z_{i_1}, \ldots, Z_{i_\ell}$ together with $d - \ell$ masks $r_{j_1}, \ldots, r_{j_{d-\ell}}$ it is sufficient to analyze the joint distribution of the ℓ variables $\overline{Z}_{i_1}, \ldots, \overline{Z}_{i_\ell}$ as defined above. Lemma 5 shows that the joint distribution of $\overline{Z}_{i_1}, \ldots, \overline{Z}_{i_\ell}$ is in fact independent of the secret variable u. Hence, an adversary that knows at most d intermediate results and masks of Algorithm 11 does not learn anything about the secret value u. Therefore, Lemma 4 together with Lemma 5 proves that Algorithm 11 is order-d perfectly masked.

Generalized Mask Changing

We can generalize securely changing of masks for intermediate results
$$u \oplus \sum_{i=1}^{l} R^{(i)} = u \oplus \sum_{i=1}^{l} \sum_{j=1}^{d} r_j^{(i)}$$
that is masked with l masks $R^{(1)}, \ldots, R^{(l)}$ each consisting of the sum of d random bytes.

Algorithm 12 Perfect Multiple Mask Change (PMMC)

Input: $Z^{(1)} = u^e \oplus R^{(1)} \oplus \ldots \oplus R^{(l)}$, $d, l \in \mathbb{N}$
$\underbrace{r_1^{(1)} \ldots, r_d^{(1)}}_{R^{(1)}}, \underbrace{r_1^{(2)}, \ldots, r_d^{(2)}}_{R^{(2)}}, \ldots, \underbrace{r_1^{(l)} \ldots, r_d^{(l)}}_{R^{(l)}}, \underbrace{r_1^{(l+1)} \ldots, r_d^{(l+1)}}_{R^{(l+1)}}$
Output: $Z_d^{(2)} = u^e \oplus R^{(l+1)}$

```
1: Z_0^(l) ← Z
2: for i = 1 ... d do
3:    Z_i^(0) ← Z_{i-1}^(l) ⊕ r_i^(l+1)        {add fresh mask r_i^(l+1)}
4:    for j = 1 ... l do
5:       Z_i^(j) ← Z_i^(j-1) ⊕ r_i^(j)         {remove old mask r_i^(j)}
6:    end for
7: end for
```

The security of Algorithm 12 can be shown using similar arguments as in the security proof of Algorithm 11.

In the sequel, we propose methods for squaring and multiplication in a secure manner.

4.5.2 Squaring

The order-d perfectly masked squaring algorithm is shown in Algorithm 13. The input $u^e \oplus R^{(1)}$ is squared in Step 1. In the following steps we use the method PMC (Algorithm 11) to change the mask $(R^{(1)})^2$ to $R^{(2)}$. Since squaring in a finite field of charactersistic 2 is a one-to-one mapping the security of the squaring step entirely relies on the security of the mask change. We showed above that the Algorithm PMC is in fact order-d perfectly masked. Hence, Algorithm 13 is also order-d perfectly masked.

Algorithm 13 Order-d Perfectly Masked Squaring (d-PMS)

Input: $x = u^e \oplus R^{(1)}, \underbrace{r_1^{(1)}, \ldots, r_d^{(1)}}_{R^{(1)}}, \underbrace{r_1^{(2)}, \ldots, r_d^{(2)}}_{R^{(2)}}$

Output: $u^{2e} \oplus R^{(2)}$

1: $f \leftarrow x^2$ $\{f = u^{2e} \oplus (R^{(1)})^2\}$

2: $t \leftarrow PMC(f, (r_1^{(1)})^2, \ldots, (r_d^{(1)})^2, r_1^{(2)}, \ldots, r_d^{(2)})$ $\{t = u^{2e} \oplus R^{(2)}\}$

4.5.3 Multiplication

The order-d perfectly masked multiplication algorithm is presented in Algorithm 14. The inputs $Z^{(1)} = u^e \oplus R^{(1)}$ and $Z^{(2)} = u^f \oplus R^{(2)}$ are multiplied in Step 1. The first loop (Steps 2-6) eliminates the first disturbing term $u^e \cdot R^{(2)}$ leaving the intermediate result

$$Z_d^{(1)} = u^{e+f} \oplus u^f \cdot R^{(1)} \oplus R^{(3)}.$$

The second disturbing term $u^f \cdot R^{(1)}$ is removed in the second loop (steps 8-12). In the final step the result is recomputed to comply to our invariant (4.3) on page 41.

We verified that Algorithm 14 is order-d perfectly masked for $d = 1, 2, 3$. Due to the large number of different distributions of the intermediate results we are not aware of an efficient method to prove the security of Algorithm 14 for arbitrary $d > 3$. We believe that Algorithm 14 is also order-d perfectly masked for $d > 3$. However, the security level provided by an order-3 perfectly masked algorithm goes far beyond the security requirements of practical applications.

Algorithm 14 Perfectly Masked Multiplication (PMM)

Input: $Z^{(1)} = u^e \oplus R^{(1)}$, $Z^{(2)} = u^f \oplus R^{(2)}$, $d \in \mathbb{N}$

$\underbrace{r_1^{(1)}, \ldots, r_d^{(1)}, r_1^{(2)}, \ldots, r_d^{(2)}}_{\text{used masks}}, \underbrace{r_1^{(3)}, \ldots, r_d^{(3)}, r_1^{(4)}, \ldots, r_d^{(4)}, r_1^{(5)}, \ldots, r_d^{(5)}}_{\text{new masks}}$

Output: $u^{e+f} \oplus R^{(5)}$

$Z_0^{(1)} \leftarrow Z^{(1)} \cdot Z^{(2)}$ $\{Z^{(1)} = u^{e+f} \oplus u^e \cdot R^{(2)} \oplus u^f \cdot R^{(1)} \oplus R^{(1)} \cdot R^{(2)}\}$

eliminate first disturbing term

for $i = 1 \ldots d$ **do**

$s_i^{(1)} \leftarrow Z^{(1)} \cdot r_i^{(2)}$ $\{s_i^{(1)} = u^e \cdot r_i^{(2)} \oplus R^{(1)} \cdot r_i^{(2)}\}$

$s_i^{(2)} \leftarrow s_i^{(1)} \oplus r_i^{(3)}$ $\{s_i^{(2)} = u^e \cdot r_i^{(2)} \oplus R^{(1)} \cdot r_i^{(2)} \oplus r_i^{(3)}\}$

$Z_i^{(1)} \leftarrow Z_{i-1}^{(1)} \oplus s_i^{(2)}$

$\{Z_i^{(1)} = u^{e+f} \oplus u^e \cdot \sum_{j=i+1}^d r^{(2)} + u^f \cdot R^{(1)} + R^{(1)} \cdot \sum_{j=i+1}^d r_i^{(2)} + \sum_{j=1}^i r_j^{(3)}\}$

end for $\{Z_d^{(1)} = u^{e+f} \oplus u^f \cdot R^{(1)} \oplus R^{(3)}\}$

eliminate second disturbing term

$Z_0^{(2)} \leftarrow Z_d^{(1)}$

for $i = 1 \ldots d$ **do**

$s_i^{(3)} \leftarrow Z^{(2)} \cdot r_i^{(1)}$ $\{s_i^{(3)} = u^f \cdot r_i^{(1)} \oplus R^{(2)} \cdot r_i^{(1)}\}$

$s_i^{(4)} \leftarrow s_i^{(3)} \oplus r_i^{(4)}$ $\{s_i^{(4)} = u^f \cdot r_i^{(1)} \oplus R^{(2)} \cdot r_i^{(1)} \oplus r_i^{(4)}\}$

$Z_i^{(2)} \leftarrow Z_{i-1}^{(2)} \oplus s_i^{(4)}$

$\{Z_i^{(2)} = u^{e+f} \oplus u^f \cdot \sum_{j=i+1}^d r^{(1)} + R^{(2)} \cdot \sum_{j=i+1}^d r_i^{(1)} + R^{(3)} + \sum_{j=1}^i r_j^{(4)}\}$

end for $\{Z_d^{(2)} \leftarrow u^{e+f} \oplus R^{(1)} R^{(2)} \oplus R^{(3)} \oplus R^{(4)}\}$

Change mask

$Z^{(3)} \leftarrow \mathrm{PMMC}(Z_d^{(2)}, (r_1^{(1)} r_1^{(2)}), \ldots, (r_d^{(1)} r_d^{(2)}), r_1^{(3)}, \ldots, r_d^{(3)}, r_1^{(4)}, \ldots, r_d^{(4)}, r_1^{(5)}, \ldots, r_d^{(5)})$

$\{Z^{(3)} \leftarrow u^{e+f} \oplus R^{(5)}\}$

4.6 Conclusions

In this chapter we analyzed the security of cryptographic algorithms such as AES against side channel attacks. We proposed a strong and general model to analyse the security. Furthermore, we proposed a generic method to implement cryptographic algorithms that is provably secure in our model. I.e., we showed that using our method, an adversary who can determine the value of a single but arbitrary intermediate result in every encryption does not derive any information about the secret key. Moreover, we analyzed the costs of our method when implemented in hardware and compared it with the efficiency of other methods. In the last part, we proposed a way to generalize our method to even more powerful adversaries that can obtain the values of an arbitrary but fixed number d of intermediate results.

Chapter 5

Fault Based Collision Attacks

In this chapter we examine the security threat caused by so called fault attacks. Fault attacks are a special type of side channel attacks in which the attacker enforces the malfunction of a cryptographic device. The output or reaction of the device is then used to derive information about the secret key. A typical target for fault attacks are smartcards (Rankl and Effing 2002). A smartcard is a general purpose computer embedded in a plastic cover of a credit card's size. The main building blocks of a smartcard are a CPU, a ROM that contains for example the operating system, an EEPROM containing among other things the secret key, and a RAM to store intermediate results of computations. To communicate with the outside world the smartcard has to be inserted into a so called *card reader* that also provides the energy the smartcard needs for operating.

Smartcards are perfectly suited for storing private information such as cryptographic keys because the corresponding cryptographic operations such as encryption or digital signature are carried out directly on the smartcard. Therefore, the key never has to leave the smartcard and hence seems to be protected very well, even in hostile environments. However, as explained in Chapter 3 (page 19) physical instances of algorithms (in hardware or software) may leak information about the computation through side channels.

(Boneh, DeMillo and Lipton 1997) were the first who showed that faults induced into the encryption process of RSA can reveal the secret key. (Biham and Shamir 1997) combined fault attacks with the concept of differentials and mounted a differential fault attack (DFA) on DES. (Skorobogatov and Anderson 2002) showed that fault attacks are realizable with sufficient precision in practice. (Blömer and Seifert 2003), (Bar-El, Choukri, Naccache, Tunstall and Whelan 2006) and (Otto 2005) give an overview of the physics of inducing faults.

In this chapter we focus on *fault attacks* on AES. The first fault attacks on AES reported in the literature were due to (Blömer and Seifert 2003) followed by improved attacks of (Dusart, Letourneux and Vivolo 2003), (Giraud 2004), (Chen and Yen 2003) and (Piret and Quisquater 2003). All these publications demonstrate the power of fault attacks. However, these attacks either use the fault model of bit resets (Blömer and Seifert 2003) in which case they do not need the faulty ciphertexts. Or the attacks only require the fault model of bit flips, in which case, however, the attacks need the faulty ciphertexts as described in (Dusart et al. 2003), (Giraud 2004), (Chen and Yen 2003), (Piret and Quisquater 2003). The fault attacks presented in this thesis use bit flips and, instead of faulty ciphertexts, the attacks only use so called collision information. This turns out to be a much weaker requirement than the requirement that an attacker gets complete faulty ciphertexts. To obtain our new attacks, we show how to combine fault attacks with so called collision attacks. In a collision attack the adversary tries to detect identical intermediate results during the encryption of different plaintexts, e.g., by using side channel information, and uses this information to derive the secret key. Basically this idea was due to Dobbertin. Schramm et al. developed collision attacks against DES (Schramm, Wollinger and Paar 2003) and AES (Schramm, Leander, Felke and Paar 2004) and showed how to detect collisions using power traces.

We combine the concepts of fault and collision attacks by inducing faults to generate collisions. This approach allows to relax the requirement of getting faulty ciphertexts to the requirement of detecting collisions in the encryption process. First we explain the basic idea underlying our attacks by presenting an attack based on some rather strong assumptions. After that we present an attack utilizing the same basic ideas that successfully attacks a smartcard that is protected by a so called memory encryption mechanism (MEM). To the best of our knowledge, this is the first fault attack on smartcards protected by memory encryption.

To defend against side channel attacks the manufacturers of smartcards developed several countermeasures. One type of countermeasure is intended to protect the card, e.g., shields, sensors or error detection. Another type is designed to render side channel attacks useless using techniques to obfuscate the side channel information, e.g. by random masking (Messerges 2000), (Golić and Tymen 2002), (Blömer et al. 2004). Yet another more efficient approach is to use a so called *memory encryption mechanism (MEM)*. Memory encryption mechanisms encrypt an intermediate result directly after it leaves the processor and decrypts

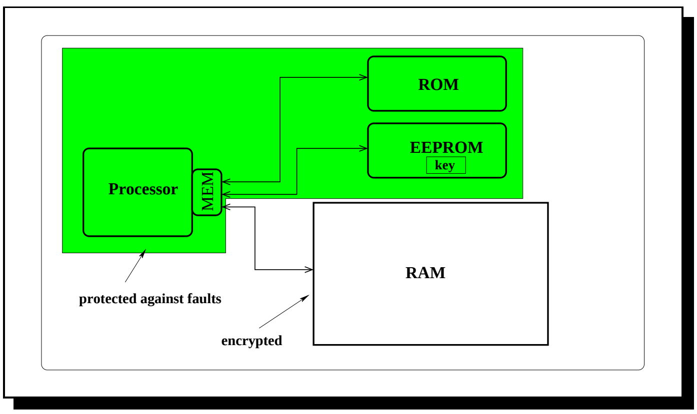

Figure 5.1: Model of an enhanced smartcard with memory encryption mechanism (MEM)

data right before it enters the processor (see Figure 5.1). This guarantees that all data stored in the RAM is encrypted. The intention is that memory encryption makes it harder for an adversary to derive information about intermediate states of the encryption process by using side channels of the smartcard. In general, it is assumed that unlike the RAM it is too difficult to induce faults into the registers of the highly integrated processor with some reasonable precision. Hence, memory encryption is widely believed to be a useful countermeasure against side channel attacks, i.e., fault attacks.

Due to the limited computational power of smartcards the MEM has to be very fast. So the manufacturers of smartcards use some light encryption algorithms that are very fast but may not be secure against serious cryptanalysis. To increase the impact of the MEM the manufacturer like to keep their algorithms secret. However, many manufacturers do not analyze the impact of MEMs on security but simply present it as an improvement of security. The strategy is to implement as many promising countermeasures as possible by not exceeding a certain cost threshold. Even a weak countermeasure should increase security.

Our attack, that works even in the presence of a MEM, shows that the security improvement of the MEM as generally used is rather limited. In particular, we present an attack on an AES implementation protected by MEM that determines the full AES key by inducing only 285 faults and detecting collisions.

The chapter is organized as follows.

Section 5.1: The Concept of Fault Attacks 52

In this section we introduce the concept of fault attacks. We categorize the existing fault attacks depending on their properties like the precision of time and location. Furthermore, we give the basic methods known so far to analyse the output or reaction of the device respectively to derive secret information.

Section 5.2: The Concept of Collision Attacks 56

To get a better understanding of fault based collision attacks we briefly sketch the idea of so called collision attacks. Later we combine this concept with fault attacks to obtain our novel concept of fault based collision attacks.

Section 5.3: New Fault Model 56

In Section 5.3 we present our model for analyzing fault based collision attacks as published in (Blömer and Krummel 2006). Fault based collsion attacks are an improvement of classical fault attacks. On one hand they do not need strong assumption like the ability to force bits to a certain value. On the other hand they do not need faulty ciphertext to derive information about the secret key. We explicitely specify the underlying assumptions and justify why fault based collision attacks are realistic threats to the security of cryptographic hardware.

Section 5.4: New Fault Attacks 59

In Section 5.4 we describe fault based collision attacks on AES and analyze their complexity. Unlike the classical fault attacks using bit flips like the attacks of (Dusart et al. 2003), (Giraud 2004), (Chen and Yen 2003) and (Piret and Quisquater 2003) obtaining faulty ciphertexts is not essential for our attacks. Therefore our attacks are applicable in scenarios where classical fault attacks do not work. On the other hand, our new attacks need more faults than the classical fault attacks. We explain the basic idea in our first attack. This attack is our basic attack and is based on rather strong assumptions. However, in the sequel we show how to strengthen it and how to adapt it to several other scenarios. The second attack we present is our strongest attack. This attack shows how to successfully attack a smartcard that is protected by a MEM. To the best of our knowledge this is the first successful attack against a smartcard protected by a MEM.

Section 5.5: Conclusions 69

We finish this chapter by reflecting the impact of fault based collision attacks on the security of recent smartcards. Furthermore, we propose some ideas of how to thwart cryptographic hardware against such attacks.

5.1 The Concept of Fault Attacks

In the sequel, we briefly summarize methods commonly suggested to induce faults in an encryption. Based on these methods we present the standard models to analyze fault attacks.

5.1.1 Methods to Induce Faults

Researchers developed a wide variety of methods to induce faults into electrical circuits. In the sequel, we list some common fault induction methods to motivate the fault models we give afterwards. The methods to induce faults are the origin to develop theoretical models for developing and analyzing fault attacks on cryptographic algorithms. Since we focus on the theoretical analysis of fault attacks we only describe each method briefly. A more complete list can be found in (Bar-El et al. 2006) and (Otto 2005).

Optical Fault Induction Exposing an electrical circuit to intensive light source will cause photoelectric effects due to the current induced by photons. In turn, these photoelectric effects cause faulty behavior of the circuit. If the circuit is laid open, intensive light is an easy way to induce faults. In (Skorobogatov and Anderson 2002) the authors showed how to induce faults with some reasonable precision using only a low-cost flash light. The precision of inducing faults this way can be improved using more sophisticated lab instruments.

Power Spikes The power supply of a smartcard is always established by the smartcard reader. To ensure that the smartcard works properly in common environments the manufacturer agreed in (ISO 2002) that a smartcard must tolerate a variation of $\pm 10\%$ of the standard supply voltage of 5V. Increasing or decreasing the voltage beyond the specified limits is called a *power spike*. Power spikes may result in a transient malfunction of the smartcard. E.g., if a power spike occurs during an encryption some intermediate operation may not work properly and produce a faulty result.

Temperature Like the supply voltage also the operating temperature of a device is restricted to certain thresholds to ensure proper operation. Heating up or cooling down a smartcard beyond these thresholds may result in malfunctions, for example modifications of the content of RAM cells.

Clock Glitches Due to the lack of an internal clock the correct operation of a smartcard entirely depends on the external clock signal that is given by the cardreader. Disturbing this clock signal may cause the card to spuriously skip operations.

X-rays and Focused Ion Beams There are two different ways to use X-rays or focused ion beams in attacking a smartcard. Firstly, they can be used to drill holes through a mechanical shield with high precision. Hence, the shield cannot prevent an attacker to access the underlying hardware with some analysis tools, e.g., a probe. Secondly, X-rays and focused ion beams can be used to induce faults without manipulating the coating of the smartcard. Details can be found in (Kömmerling and Kuhn 1999).

Eddy Current The French physicist Leon Foucault discovered in 1851 that moving a conductor through a magnetic field causes some current flow called eddy current. Using eddy current to disturb the operations of a smartcard is one of the oldest proposed methods of fault induction. See for example (Kocher 1996), (Anderson and Kuhn 1996) and (Kömmerling and Kuhn 1999). However, it is difficult to focus the fault to a certain area of the chip. In (Quisquater and Samyde 2002) the authors developed a refined method of inducing eddy current.

5.1.2 Fault Models

To analyze fault based attacks we first have to develop adequate models that cover the important aspects of real environments. Independent of the method to induce faults the following properties are essential for the analysis of fault attacks:

Precision The precision of the fault induction is crucial for both the success and the complexity of a fault based attack. We distinguish between the precision of time and location. The precision of location defines the ability of the attacker to focus the fault induction on a certain part of the hardware. To induce a fault into a specific intermediate result an adversary must also be able to induce faults at a precise time depending on the progress of the encryption.

Number of affected bits This property specifies how many bits are affected by the induced faults. Precise fault injection techniques can modify single bits whereas other techniques may change bytes or even a whole intermediate state.

Effect of the fault Our strongest model allows the adversary to set a bit of an intermediate result to a certain value. I.e., if an adversary $\mathcal{A}$ can force a bit to be 0 we call this a *bit reset*. Moreover, if $\mathcal{A}$ can force a bit to be 1 we call it a *bit set*. In weaker models the adversary does not have such a strong influence on the value of faulty intermediate result. E.g., if the adversary can only modify a whole intermediate state it is very unlikely that he can force the complete state to a chosen value. In such scenarios we assume that he can change the intermediate state in a random and unpredictable way. We call this *random fault*.

Incidence of fault The incidence of a fault also plays an important role in the analysis of fault based attacks. A fault that only changes the content of a memory cell once and works properly during the rest of the encryption is called a *transient fault*. For example, a focused ion beam changes the content of some bits in the RAM but does not destroy any transistor. In contrast *permanent faults* are defective parts of the hardware that do not work correctly after the fault is induced. E.g., this could be an interrupted wire that prevents the information flow.

A fault attack can be divided into two steps. In the first step the adversary $\mathcal{A}$ induces a fault into the encryption, e.g., by using a method described above. We call this step *fault induction step.*

In the second step of a fault based attack, $\mathcal{A}$ analyses the impact of the induced fault on the encryption. Depending on the implementation and the abilities of $\mathcal{A}$ the analysis differs. We distinguish two kinds of fault based attacks:

Fault Attacks Based on the Analysis of Faulty Output

If the encryption algorithm is not protected against fault attacks it does not react on the fault directly. It simply continues its computation based on faulty intermediate results and outputs a faulty ciphertext in the end. Giving $\mathcal{A}$ access to both the faulty and the corresponding correct ciphertext allows him to backtrack the encryption and deduce information about the last round key. E.g., for ciphers like AES an adversary $\mathcal{A}$ performs the following procedure:

1. $\mathcal{A}$ guesses the i-th byte $\widehat{k}_i^{10}$ of the last round key and computes some intermediate results by tracing back the last round of the encryption for both the faulty and the correct ciphertext.

2. $\mathcal{A}$ verifies whether the difference of the corresponding intermediate results could be caused by the induced fault. If this difference could not be caused by the fault, the

candidate $\widehat{k}_i^{10}$ is proven to be wrong and discarded. In the other case, $\mathcal{A}$ keeps $\widehat{k}_i^{10}$ as a possible key value.

See for example (Dusart et al. 2003) and (Giraud 2004) for fault attacks of this type on AES.

Fault Attacks based on the Information whether the Output is Faulty or Not

If the implementation does not output the faulty ciphertext the analysis is more involved. However, we assume that the adversary $\mathcal{A}$ always notices if the induced fault falsifies the encryption. E.g., a so called security reset that puts the implementation into a specified state after detecting a fault would reveal the information that a fault occurred. But even if the implementation does not reveal the detection of a fault directly it has to react on the fault somehow. For example, it might recompute the encryption. But such a special treatment of faults could be detected by the adversary by simply measuring the time an encryption takes. If a faulty encryption takes longer than the correct encryption the induced fault had an impact on the encryption. If the faulty encryption is as fast as the correct encryption the adversary concludes that the induced fault does not influence the encryption. We distinguish two kinds of attacks:

try and error attack The so called *try and error attack* works if $\mathcal{A}$ can set or reset bits. After forcing a bit of an intermediate result to either 0 (or 1), $\mathcal{A}$ determines if a fault occurred or not. If the encryption is correct then the fault attack did not change the value of that bit. Hence, $\mathcal{A}$ concludes that the bit was 0 (or 1 respectively). If the encryption is faulty then the fault attack changed the value of that bit. Hence, $\mathcal{A}$ concludes that the bit was 1 (or 0 respectively). Repeating fault attacks $\mathcal{A}$ can determine the values of several bits of an intermediate state. In turn, $\mathcal{A}$ can use this information to derive information about the secret key. See for example (Blömer and Seifert 2003) for this kind of fault attacks on AES.

fail safe attack Like the try and error attack, the so called *fail safe attack* does not need a faulty ciphertext either but also works with random faults. To illustrate the attack consider the square and multiply always algorithm (Algorithm 15) to compute an RSA signature.

This implementation was proposed to counteract side channel attacks like timing and power analysis. However, it turned out that this implementation is susceptible to a fail safe attack. The idea is as follows: The attacker induces a random fault into t_1 of the jth execution of the loop in line 4 of Algorithm 15. He can then determine the jth bit d_j of the secret exponent as follows. If $d_j = 0$ then no multiplication is needed in step j to compute the signature and the variable t_1 does not influence the computation. Hence the result would be correct. If $d_j = 1$ then t_1 is needed in step j to compute

---

Algorithm 15 square and multiply always

Input: RSA modulus N, message $m \in \mathbb{Z}_N$, secret exponent $d = \sum_{i=0}^{\ell-1} d_i \cdot 2^i \in \mathbb{Z}^*_{\varphi(N)}$, $d_i \in \{0,1\}$
Output: $m^d \mod N$
```
t ← 1
for i = l − 1 to 0 do
    t_0 ← t^2
    t_1 ← t_0 · m                                  {multiply always}
    if d_i = 0 then
        t ← t_0
    else
        t ← t_1
    end if
end for
```

---

the signature and hence the result would be incorrect. $\mathcal{A}$ can determine the complete secret key by repeating this attack for all bits of d.

5.2 The Concept of Collision Attacks

The idea of collision attacks was due to Dobbertin and the first collision attacks were published in (Schramm et al. 2003) and (Schramm et al. 2004). A *collision* is the occurrence of identical intermediate results in the encryptions of different plaintexts. An adversary $\mathcal{A}$ tries to detect collisions and uses this information together with the plaintexts (or ciphertexts) to derive information about the secret key. To detect collisions the authors of (Schramm et al. 2003) and (Schramm et al. 2004) proposed to use side channel information, e.g. power traces, mounted successful collision attacks on DES and AES.

5.3 New Fault Model

5.3.1 Notation

In this chapter we focus on the AES-128 symmetric cipher and simply call it AES. However, the attacks presented in this chapter can also be easily adapted to other versions of AES having larger key sizes. As defined in Chapter 2, let $\mathcal{P} := \{0,1\}^{128}$ be the set of plaintexts, $\mathcal{C} := \{0,1\}^{128}$ be the set of ciphertexts and $\mathcal{K} := \{0,1\}^{128}$ be the set of keys. In the classical model the AES encryption with a fixed key $k \in \mathcal{K}$ is a bijective function :

$$\begin{aligned} \mathrm{AES}_k : \mathcal{P} &\rightarrow \mathcal{C} \\ p &\mapsto c := \mathrm{AES}_k(p). \end{aligned}$$

Let

$$O := \{\mathtt{SB}, \mathtt{SR}, \mathtt{MC}, \mathtt{AR}\}$$

denote the set of round transformations `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`. Furthermore, let

$$p_{i,j}^{(r),(o)}$$

denote the jth bit of the ith byte of the encryption state of plaintext p after the operation $o \in O$ of round $1 \leq r \leq 10$. For example, $p_{5,3}^{(3),(\mathtt{SR})}$ is the 3rd bit of the 5th byte of the encryption state of plaintext p after the `ShiftRows` transformation of round 3. In the sequel, we will omit the index j that defines the bit position if it is not relevant in that context. The ith byte of the round key $k^{(r)}$ of round r is called $k_i^{(r)}$.

We can then define the set of bits that are results of a round transformation as

$$\begin{aligned} \mathcal{S} \quad := \quad & \left\{p_{i,j}^{(0),(\mathtt{AR})} \mid \; i \in \{0,\ldots,15\}, \; j \in \{0,\ldots,7\}\right\} \cup \\ & \left\{p_{i,j}^{(r),(o)} \mid o \in O, \; r \in \{1,\ldots,10\}, \; i \in \{0,\ldots,15\}, \; j \in \{0,\ldots,7\}\right\}. \end{aligned}$$

5.3.2 Model

To model faults mathematically, we extend the AES function with a second variable b that specifies a bit position during the computation of AES_k. The set of all realizable functions via AES is extended by flipping bit b during the computation of AES_k. We call the extended function FAES:

$$\begin{aligned} \mathrm{FAES}_k : \mathcal{P} \times \mathcal{S} \quad &\rightarrow \quad \mathcal{C} \\ (p, b) \quad &\mapsto \quad c := \mathrm{FAES}_k(p, b) \end{aligned}$$

However, the extended function called $\mathrm{FAES}(p, b)$ is not bijective. There exist collisions such that two intermediate states of computations of $\mathrm{FAES}(p, b)$ and $\mathrm{FAES}(p', b')$ with different inputs $(p, b) \neq (p', b')$ are equal. An attacker wants to detect those collisions and then use them to derive the secret key k.

In our scenario we have a smartcard with an implementation of AES and a secret AES key k stored on it. An adversary $\mathcal{A}$ has access to the smartcard and wants to determine information about the secret key k. In our model we assume that the following holds:

1. $\mathcal{A}$ is able to trigger the smartcard to encrypt chosen plaintexts.
2. $\mathcal{A}$ can induce transient bit flips into the encryption process.
3. $\mathcal{A}$ is able to detect collisions.

Discussion of the New Model

To simplify the description of our attacks we assume that the adversary $\mathcal{A}$ is able to input chosen plaintexts into the smartcard. However, our attacks can also be transformed to known plaintext attacks. During the encryption $\mathcal{A}$ can induce faults in terms of transient bit flips into the result of a round transformation that is stored in the RAM. To be more precise, $\mathcal{A}$ can flip a single bit of some specified byte in the memory. Furthermore, $\mathcal{A}$ can detect collisions by obtaining some information about an internal state of the encryption process. However, we do not assume that this information lets $\mathcal{A}$ determine (parts of) the secret key directly. Nevertheless, it enables $\mathcal{A}$ to detect if a collision occurred or not. We call any kind of information that lets $\mathcal{A}$ detect collisions *collision information* of some intermediate state of $\text{FAES}(p, b)$. Later we will show examples how to derive collision information.

Modeling Collision Information We model collision information as the evaluation of an injective function f_k that depends on the concrete implementation of AES_k and the secret key k. It gets as input the specification of a bit position that is flipped during the encryption of a plaintext p. The output is some information about an intermediate state of the encryption. According to the notation introduced above we denote the collision information of encrypting plaintext p and inducing a bit flip of bit e of byte i of the state after transformation o in round r by $f_k(p_i^{(r),(o)}, e)$. E.g., $f_k(p_i^{(1),(\texttt{SB})}, e)$ is the collision information $\mathcal{A}$ obtains after flipping bit e of byte i of the state after the `SubBytes` transformation of round 1. It is also possible to derive the collision information without inducing a fault. We denote the evaluation of f_k without inducing a fault in the encryption process by $f_k(p_i^{(r),(o)}, -)$.

Realizations of Collision Information Depending on the purpose of the smartcard f_k can have different realizations. Given the ciphertexts the detection of collisions is easy because the equality of ciphertexts implies equality of intermediate states. However, in many cases the output of an encryption is not available to the attacker. For example, if the smartcard computes a message authentication code (MAC) or a hash value using AES as a building block, f_k can simply be the MAC or the hash value. Remember that the MAC is the final result of a number of interlinked AES encryptions and not the result of a single AES encryption. The final ciphertext could also be used as collision information if the smartcard computes multiple encryption with different encryption algorithms. Finally, if the smartcard computes a single encryption but does not output faulty ciphertexts, f_k could be the measurement of some side channel information, e.g., power trace, that allows to detect collisions.

Cost Analysis To analyze the costs of a fault based collision attack we simply count the number of faults we have to induce. The evaluation of f_k without inducing a fault is for free. We also neglect the complexity of additional computations that can be performed offline since in our cases they are obviously easy.

5.4 Fault Based Collision Attacks on AES

Now we describe fault based collision attacks on AES. For simplicity, we only show how to compute byte k_0 of the secret key k. Similar approaches can be used to compute the other key bytes.

We describe how to mount and analyze fault based collision attacks on AES in different scenarios. Each scenario is characterized by abilities of the adversary and the properties of the environment.

Precision of Fault Induction The first characteristic defines the precision of the fault induction. We consider two cases. In the first case the adversary $\mathcal{A}$ is able to flip a specific bit of an intermediate state. In the second case the adversary can focus the fault to a single byte of an intermediate state. However, we assume that he cannot focus on a single bit of that byte but each possible bit flip occurs with probability 1/8.

Memory Encryption Mechanism (MEM) The second characteristic specifies whether the smartcard is protected by a MEM or not. The MEM encrypts every intermediate result that leaves the processor and decrypts a value right before it enters the processor, see Figure 5.1 (page 50). Since a smartcard has only restricted computational power and memory most manufacturers choose a byte oriented encryption function with a fixed key that is used for encryption and decryption. In our approach we simply model the memory encryption as an unknown but fixed function $h : \{0,1\}^8 \to \{0,1\}^8$. That means that we do not rely on a weakness in the memory encryption itself. In particular, we do not assume to have any information of how bit flips affect further processing of that byte.

Validation of Collision Information The last characteristic defines whether collision information remains valid for a long period of time or not. If collision information does not remain valid there is no reason for $\mathcal{A}$ to store collision information since he cannot use it later in the attack. $\mathcal{A}$ is only able to compare collision information of two recently taken measurements and store the result. This effect could be caused by environments that are frequently changed such that collision information taken at different times is hardly comparable, e.g., due to some countermeasure that induces noise into the collision information. If, however, collision information remains valid over the time span used for the attack it may be useful for $\mathcal{A}$ to store this information in a preprocessing step to have it available once and for all. It will turn out later that stored information helps to reduce the number of induced faults.

We denote the transformation of `SubBytes` applied on a single byte x of the state simply as the application of the sbox on x and write it as $\mathbf{S}[x]$. To simplify notation we define

$$\Delta(p_i, q_i) = p_i \oplus q_i$$

to be the difference of two plaintext bytes p_i and q_i. Then

$$\Delta_{in}(p_i, q_i) = (p_i \oplus k_i^{(0)}) \oplus (q_i \oplus k_i^{(0)}) = p_i \oplus q_i$$

is the input difference of (p_i, q_i) before the first application of the sbox and

$$\Delta_{out}(p_i, q_i) = \mathbf{S}[p_i \oplus k_i^{(0)}] \oplus \mathbf{S}[p_i \oplus k_i^{(0)}]$$

is the output difference of (p_i, q_i) after the first application of the sbox.

5.4.1 Basic Attack

First, we describe the scenario in which the attack takes place. We assume that $\mathcal{A}$ can flip a specific bit at position e of the intermediate state $p^{(1),(\mathtt{SB})}$. We also assume that collision information remains valid over the time span of the attack. Finally, we assume that the smartcard is not protected by a MEM.

In a preprocessing step the adversary computes an array B_e of length 256. In position $B_e[y], y \in \{0, \ldots, 255\}$ the array stores the following information:

$$B_e[y] := \big\{\{s, t\} \,\big|\, s \oplus t = y, \mathbf{S}[s] \oplus \mathbf{S}[t] = 2^e\big\},$$

i.e., $B_e[y]$ stores all (unordered) pairs of bytes with $\Delta_{in}(s, t) = y$ and

$$\Delta_{out}(s, t) = \mathbf{S}[s] \oplus \mathbf{S}[t] = 2^e.$$

Furthermore, by $C_e[y]$ denote the union of sets in $B_e[y]$. The sets $C_e[y]$ are pairwise disjoint. As it turns out, for every $e \in \{0, 1, \ldots, 7\}$ we have that 129 sets $C_e[y]$ are empty, 126 sets $C_e[y]$ contain exactly two elements, and one set $C_e[y]$ contains exactly four elements.

Next, $\mathcal{A}$ collects a set B of collision information $f_k(p_0^{(1)(\mathtt{SB})}, -)$ for all 256 different values of p_0 and arbitrary but fixed $p_1, \ldots, p_{15}$. Then $\mathcal{A}$ chooses an arbitrary value q_0 and encrypts the corresponding plaintext flipping an arbitrary bit e of $q_0^{(1),(\mathtt{SB})}$. If f_k has the property that

$$f_k(p_0^{(1),(\mathtt{SB})}, -) = f_k(q_0^{(1),(\mathtt{SB})}, e)$$

then $\mathcal{A}$ is able to find the corresponding plaintext p_0 satisfying

$$\mathbf{S}[p_0 \oplus k_0] = \mathbf{S}[q_0 \oplus k_0] \oplus 2^e$$

by comparing the collision information with the elements of B. Given the pair p_0, q_0 the adversary $\mathcal{A}$ knows the difference $p_0 \oplus k_0 \oplus q_0 \oplus k_0 = p_0 \oplus q_0$. Using array B_e the adversary $\mathcal{A}$ now concludes

$$\{p_0 \oplus k_0, q_0 \oplus k_0\} \in B_e[p_0 \oplus q_0].$$

Hence, $\mathcal{A}$ knows that the correct key byte k_0 satisfies

$$k_0 \in \big\{p_0 \oplus s \,\big|\, s \in C_e[p_0 \oplus q_0]\big\}. \tag{5.1}$$

As mentioned above, $\left|C_e[y]\right| \leq 4$ for all y, and $\left|C_e[y]\right| = 2$ for all but one y. Hence, at this point $\mathcal{A}$ has reduced the number of possible values for key byte k_0 to at most 4.

Next, $\mathcal{A}$ repeats the experiment described above with some value q_0', such that $q_0' \oplus s \notin C_e[p_0 \oplus q_0]$ for all $s \in \{p_0 \oplus \bar{s} \mid \bar{s} \in C_e[p_0 \oplus q_0]\}$. Using the collision information in set B, the adversary $\mathcal{A}$ determines p_0' such that $\mathbf{S}[p_0' \oplus k_0] = \mathbf{S}[q_0' \oplus k_0] \oplus 2^e$. As before $\mathcal{A}$ concludes that the key byte k_0 satisfies

$$k_0 \in \left\{p_0' \oplus s \,\middle|\, s \in C_e[p_0' \oplus q_0']\right\}. \tag{5.2}$$

By choice of q_0', the adversary $\mathcal{A}$ is guaranteed that $p_0 \oplus q_0 \neq p_0' \oplus q_0'$. By elementary arithmetic it follows that if $\left|C_e[p_0' \oplus q_0']\right| = \left|C_e[p_0 \oplus q_0]\right| = 2$, then (5.1) and (5.2) uniquely determine the key byte k_0. By analyzing the structure of the arrays B_e we verified that the key byte k_0 is also uniquely determined if one of the sets has size four.

Cost Analysis To determine a single AES key byte $\mathcal{A}$ has to induce two faults. Thus 32 faults are enough to determine the complete 128-bit AES key.

5.4.2 Second Attack

The scenario for this attack is as follows. We assume that the adversary $\mathcal{A}$ can flip a specific bit e of the intermediate state $p^{(0),(\mathtt{AR})}$. We also assume that collision information remains valid over the time span of the attack. Finally, we assume that the smartcard is protected by a MEM modelled as a function $h : \{0,1\}^8 \to \{0,1\}^8$. This implies that after a flip of bit e the encryption continues using the value $h^{-1}(h(p_i \oplus k_i) \oplus 2^e)$ instead of $p_i \oplus k_i$. Therefore, we assume that we have no information about the impact of bit flips on the encryption process.

The attack is divided into two steps. In the first step the adversary $\mathcal{A}$ collects the necessary information to compute a function g_0 that is equal to h up to some constant coefficient. To do so $\mathcal{A}$ selects a set S of 256 plaintexts p that take on all different values in byte p_0 and that are equal in each other byte. $\mathcal{A}$ uses the smartcard to derive the collision information for each of these plaintexts by evaluating $f_k(h(p_0^{(0)(\mathtt{AR})}), -)$ and stores it in the table B. Then $\mathcal{A}$ encrypts plaintexts p of the set S and induces a bit fault into bit $0 \leq e \leq 7$ of $h(p_0^{(0),(\mathtt{AR})})$ and compares the collision information $f_k(h(p_0^{(0),(\mathtt{AR})}), e)$ with the entries of table B to find the corresponding plaintext p_0'. So $\mathcal{A}$ knows the difference

$$h(p_0 \oplus k_0) \oplus h(p_0' \oplus k_0) = 2^e$$

and stores the triple (p_0, p_0', e) in a difference table ΔB. This step is repeated for different plaintexts p and for different faulty bit positions until $\mathcal{A}$ received enough information to compute the differences

$$h(p_0 \oplus k_0) \oplus h(p_0' \oplus k_0)$$

of one byte p_0 with all other bytes p_0'. The details are given in the following lemma.

Lemma 6 *Let $m : \{0,1\}^q \to \{0,1\}^q$ be an unknown function defined over $\mathbb{F}_{2^q}$. There exists a set D of $2^q - 1$ pairs $(u, v) \in \mathbb{F}_{2^q} \times \mathbb{F}_{2^q}$ with the following property: If for all $(u, v) \in D$ we know $e \in \{0, \ldots, q-1\}$ such that $m(u) \oplus m(v) = 2^e$, then one can determine a function $g : \{0,1\}^q \to \{0,1\}^q$ such that $g \oplus c = m$ for some constant $c \in \mathbb{F}_{2^q}$.*

Proof. Given some set $D \subseteq \mathbb{F}_{2^q} \times \mathbb{F}_{2^q}$ we construct a graph G whose set of vertices is $\mathbb{F}_{2^q}$ as follows. We connect two vertices u, v with an edge of weight e if $(u, v) \in D$.

If in G there exists a path between two vertices x, y then the difference $m(x) \oplus m(y)$ is determined by the differences of pairs in D. Furthermore, if the graph G is connected we can compute the difference $m(x) \oplus m(y)$ for all $(x, y) \in \mathbb{F}_{2^q} \times \mathbb{F}_{2^q}$. In particular, we can determine all differences of the form $m(u) \oplus m(u_0)$ for an arbitrary but fixed input u_0. Using Lagrange interpolation we can compute the function $g(u) = m(u) \oplus m(u_0)$. Setting $c := m(u_0)$ proves the lemma.

Next we describe a set D of pairs (u, v) with known differences $m(u) \oplus m(v) = 2^e$, such that the graph G as defined above is in fact connected. First we fix an arbitrary $e_1 \in \{0, \ldots, q-1\}$. Then there exists a set D_1 of 2^{q-1} distinct pairs $(u, v) \in \mathbb{F}_{2^q} \times \mathbb{F}_{2^q}$ such that $m(u) \oplus m(v) = 2^{e_1}$. All pairs in D_1 will be elements of D. If we consider the graph whose edges are defined by pairs in D_1 we get a graph G_1 on the vertex set $\mathbb{F}_{2^q}$ that consists of 2^{q-1} connected components each consisting of exactly 2 vertices.

Next we choose $e_2 \neq e_1$. Then there exists a set D_2 of 2^{q-2} pairs of vertices (u, v) with $m(u) \oplus m(v) = 2^{e_2}$ such that each pair in D_2 connects different connected components of G_1. We call the resulting graph G_2. The set D will also contain all elements from D_2.

Continuing in this way with all possible $e_i \in \{0, \ldots, q-1\}$ we get sets of pairs $D_1, D_2 \ldots, D_q$ and graphs $G_1, G_2, \ldots, G_q$ such that G_i has 2^{q-i} connected components. In particular, G_q is connected. Moreover, the edges of G_q are given by the pairs in $D := \bigcup_{i=1}^{q} D_i$. The size of D is $2^q - 1$. This proves the lemma. $\square$

We want to apply Lemma 6 to the function $h(x \oplus k_0)$. It is easy to see that $\mathcal{A}$ can compute exactly the set of differences D described in the proof of Lemma 6 since he is able to flip a specific bit. Hence, knowing D the adversary $\mathcal{A}$ can compute a function $g_0 : \{0,1\}^8 \to \{0,1\}^8$ such that for all $x \in \mathbb{F}_{256}$ the difference $g_0(x) \oplus h(x \oplus k_0)$ is some constant $c_0 \in \mathbb{F}_{256}$. Since $\mathcal{A}$ does not know the constant c_0 he does not get any information about the key byte k_0 at this point.

$\mathcal{A}$ continues by computing for all other byte positions $1 \leq i \leq 15$ functions $g_1, \ldots, g_{15}$ such that for all $x \in \mathbb{F}_{256}$ the function $g_i : \{0,1\}^8 \to \{0,1\}^8$ has the property that $g_i(x) \oplus h(x \oplus k_i) = c_i$ for some unknown constant $c_i \in \mathbb{F}_{256}$. Each of the g_i's does not reveal any information about the involved key byte k_i because the constant c_i can take on all possible values and is unknown to $\mathcal{A}$.

To derive information about the key, $\mathcal{A}$ proceeds as follows. He guesses two candidates

$\widehat{k}_0, \widehat{k}_i$ for the keybytes k_0, k_i, respectively. To test this hypothesis on the key, $\mathcal{A}$ selects several bytes x uniformly at random and computes

$$g_0(x \oplus \widehat{k}_0) = h(x \oplus \widehat{k}_0 \oplus k_0) \oplus c_0$$

and

$$g_i(x \oplus \widehat{k}_i) = h(x \oplus \widehat{k}_i \oplus k_i) \oplus c_i.$$

Depending on the hypothesis $(\widehat{k}_0, \widehat{k}_i)$ the difference

$$t_{0,i} := g_0(x \oplus \widehat{k}_0) \oplus g_i(x \oplus \widehat{k}_i)$$

computes to

$$h(x) \oplus c_0 \oplus h(x) \oplus c_i = c_0 \oplus c_i \quad \text{, if} \quad \widehat{k}_0 \oplus k_0 = \widehat{k}_i \oplus k_i \tag{5.3}$$
$$h(x \oplus \widehat{k}_0 \oplus k_0) \oplus c_0 \oplus h(x \oplus \widehat{k}_i \oplus k_i) \oplus c_i \quad \text{, if} \quad \widehat{k}_0 \neq k_0 \text{ and } \widehat{k}_i \neq k_i \tag{5.4}$$
$$h(x) \oplus c_0 \oplus h(x \oplus \widehat{k}_i \oplus k_i) \oplus c_i \quad \text{, if} \quad \widehat{k}_0 = k_0 \text{ and } \widehat{k}_i \neq k_i \tag{5.5}$$
$$h(x \oplus \widehat{k}_0 \oplus k_0) \oplus c_0 \oplus h(x) \oplus c_i \quad \text{, if} \quad \widehat{k}_0 \neq k_0 \text{ and } \widehat{k}_i = k_i \tag{5.6}$$

Now we assume that the function h has the following property. There do not exist constants $a, c \in \mathbb{F}_{256}$ such that $h(x) \oplus a = h(x \oplus c)$ for all x. Note that this assumption does not restrict the choice of h for two reasons. Firstly, a function used for memory encryption that does not have this property contains too much structure and is probably easier to attack. Secondly, most functions have this property. In fact, a random function has the property with probability at least $1 - 2^{-127}$.

This assumption implies that unlike in case (5.3) in cases (5.4),(5.5),(5.6) the difference $t_{0,i}$ is not constant. Moreover, if the guess $\widehat{k}_0, \widehat{k}_i$ was correct that is $\widehat{k}_0 = k_0$ and $\widehat{k}_i = k_i$ then $\mathcal{A}$ will always be in case (5.3). Now $\mathcal{A}$ can easily test the hypothesis $(\widehat{k}_0, \widehat{k}_1)$ by computing $t_{0,i}$ for several bytes x. If $t_{0,i}$ varies for several different values of x then $\mathcal{A}$ knows that he is not in case (5.3). It follows that the pair $(\widehat{k}_0, \widehat{k}_1)$ cannot be correct. On the other hand if $t_{0,i}$ remains constant $\mathcal{A}$ concludes to be in case (5.3) and keeps the pair $(\widehat{k}_0, \widehat{k}_1)$ as a potentially correct candidate.

This implies that for every possible key byte $\widehat{k}_0$ the adversary $\mathcal{A}$ obtains a single candidate $\widehat{k}_i$ for $1 \leq i \leq 15$ that fulfills condition (5.3). Guessing $\widehat{k}_0$ the adversary $\mathcal{A}$ can compute a vector $(\widehat{k}_1, \ldots, \widehat{k}_{15})$ composed of unique candidates $\widehat{k}_i$ that only depend on $\widehat{k}_0$. To uniquely determine the correct key, $\mathcal{A}$ simply mounts an exhaustive search attack on the 256 possible values of $\widehat{k}_0$.

Cost Analysis $\mathcal{A}$ has to induce 255 faults to compute a function g_i according to Lemma 6. To test a hypothesis of the key $\mathcal{A}$ does not need to induce faults. So the overall number of faults is $16 \cdot 255 = 4080$.

Improvement The previous attack can be improved with respect to the number of induced faults as shown below. In the first step $\mathcal{A}$ computes the function g_0 such that $g_0(x) = h(x \oplus k_0) \oplus c_0$, where $c_0 \in \mathbb{F}_{256}$ is unknown, as above. To determine the other functions $g_1, \ldots, g_{15}$ the adversary $\mathcal{A}$ uses the fact that each g_i is related to g_0 by the following equation

$$g_i(x) = h(x \oplus k_i) \oplus c_i = g_0(x \oplus \underbrace{k_i \oplus k_0}_{s_i}) \oplus c_i \oplus c_0.$$

So knowing g_0 (determined as above) $\mathcal{A}$ computes a list of all 256 functions $g_{0,s} := g_0(x \oplus s)$, $s \in \mathbb{F}_{256}$. To determine which of these functions equals g_i the adversary $\mathcal{A}$ chooses arbitrary p_i, q_i and evaluates $f_k(h(p_i^{(0),(\mathtt{AR})}), -)$ and $f_k(h(q_i^{(0),(\mathtt{AR})}), e)$ at byte position i. Using this information $\mathcal{A}$ computes some differences $g_i(p_i) \oplus g_i(q_i)$ as described in the computation of g_0 above.

To determine the correct function $g_i = g_{0,s_i}$, the adversary $\mathcal{A}$ simply checks which of the functions $g_{0,s}$ fulfills these differences simultaneously until only one function remains. See below for the required number of experiments. Then $\mathcal{A}$ knows the sum $s_i = k_0 \oplus k_i$ of two AES key bytes. $\mathcal{A}$ repeats this procedure for all other byte positions $0 \leq i \leq 15$. As before guessing $\widehat{k}_0$ the adversary $\mathcal{A}$ can determine a unique candidate $\widehat{k}_i$. That means that $\mathcal{A}$ has a vector $(\widehat{k}_1, \ldots, \widehat{k}_{15})$ with fixed candidates $\widehat{k}_i$ for each of the 256 candidates $\widehat{k}_0$. Like in the original version of this attack this reduces the set of possible AES keys to only 256 candidates. An exhaustive search reveals the full AES key.

Cost Analysis To compute g_0 the adversary $\mathcal{A}$ has to induce 255 faults like in the original version. To determine further g_i's, $\mathcal{A}$ has to collect a set of differences $g_i(p) \oplus g_i(q)$ that is fulfilled by only one of the 256 functions $g_{0,s}$ simultaneously. Notice that if the function $g_{0,s}$ fulfills a difference, i.e., $g_0(p \oplus s) \oplus g_0(q \oplus s) = g_i(p) \oplus g_i(q)$ then because of symmetry the function $g_{0,s'}$ given by $s' := p \oplus q \oplus s$ also fulfills this difference since

$$g_0(p \oplus (p \oplus q \oplus s)) \oplus g_0(q \oplus (p \oplus q \oplus s) = g_0(q \oplus s) \oplus g_0(p \oplus s) = g_i(q) \oplus g_i(p).$$

Assuming that the 256 functions $g_{0,s}$ behave like random permutations (except for the symmetry) we expect that $\mathcal{A}$ needs 2 differences to uniquely identify the correct one with high probability. We tested this assumption by various experiments and in our experiments it proved to be correct. Hence, we expect that $\mathcal{A}$ needs $255 + 15 \cdot 2 = 285$ faults to determine the full 128-bit AES key.

As mentioned before we do not consider the complexity of the offline computations like Lagrange interpolation etc. since all these computations can be performed efficiently without access to the smartcard.

5.4.3 Third Attack

First, we describe the scenario in which the attack takes place. We assume that $\mathcal{A}$ can flip a specific bit at position e of the intermediate state $p^{(1),(\mathtt{SB})}$. We do not assume that collision information remains valid over the time span of the attack. Hence, $\mathcal{A}$ is only able to compare collision information of two recently obtained measurements. Finally, we assume that the smartcard is not protected by a MEM. Because it is always clear from the context we simplify notation by identifying elements of $\mathbb{F}_{256}$ with their canonical representation as elements of the set $\{0, \ldots, 255\}$.

As a basis for his attack $\mathcal{A}$ fixes some input difference Δ_{in} and output difference Δ_{out} of the application of the sbox in round 1. To be able to detect collisions with a single bit flip we restrict Δ_{out} to be a power of 2.

The analysis of the sbox shows that there are a lot of suitable values for Δ_{in} and Δ_{out} . E.g., $\mathcal{A}$ chooses $\Delta_{in} = 10$ and $\Delta_{out} = 4$. Only the two pairs

$$Z_1 := (p_0 \oplus k_0 = 0, q_0 \oplus k_0 = 10)$$

and

$$Z_2 := (p_0 \oplus k_0 = 244, q_0 \oplus k_0 = 254)$$

together with their commuted counterparts fulfill the chosen requirements. A fault that is induced into bit 2 of $q_0^{(1),(\texttt{SB})}$ after the application of the sbox results in a collision for one of these pairs. In order to detect such a collision the collision information f_k should have the property that

$$f_k(p_0^{(1),(\texttt{SB})}, -) = f_k(q_0^{(1),(\texttt{SB})}, 2).$$

If $\mathcal{A}$ finds such a collision he can conclude that the key byte k_0 is an element of the set

$$\{p_0 \oplus 0, p_0 \oplus 10, p_0 \oplus 244, p_0 \oplus 254\}.$$

More precisely, the attack using f_k with the property defined above works as follows. First, $\mathcal{A}$ generates all 128 pairs of plaintexts (p, q) (without symmetry) that have difference 10 in byte 0 ($p_0 = q_0 \oplus 10$) and are equal in the other bytes, i.e.,

$$\Delta(p_i, q_i) = \begin{cases} 10, \text{ if i=0} \\ 0, \text{ otherwise.} \end{cases}$$

$\mathcal{A}$ knows that exactly two of these pairs have output difference 4 in byte 0. The input difference of the sbox is the same as the difference of p_0 and q_0 since `AddRoundKey` does not change it. $\mathcal{A}$ checks all 128 pairs (p, q) until

$$f_k(p_0^{(1),(\texttt{SB})}, -) = f_k(q_0^{(1),(\texttt{SB})}, 2).$$

Taking the symmetry into account it follows that either $p_0 \oplus k_0 = 0$, $p_0 \oplus k_0 = 10$, $p_0 \oplus k_0 = 244$ or $p_0 \oplus k_0 = 254$. So there are only 4 candidates for k_0 left. $\mathcal{A}$ can repeat this attack for all byte positions of the state. This leaves $2^{2 \cdot 16} = 2^{32}$ possible keys. To determine the complete 128-bit AES key $\mathcal{A}$ mounts an exhaustive search attack.

Cost Analysis In the first step $\mathcal{A}$ examines 128 pairs of plaintexts with difference 10. Two of these pairs result in a collision so the expected number of faults $\mathcal{A}$ has to induce is $(2/128)^{-1} = 64$. To compute a 128 bit AES key, $\mathcal{A}$ expects to induce $16 * 64 = 1024$ faults and a brute force attack of size 2^{32}.

Alternative To determine the correct candidate of the key byte $\mathcal{A}$ could also repeat the same procedure as above with another difference. We assume that f_k lets $\mathcal{A}$ detect collisions when flipping bit 3, i.e.

$$f_k({p'}_0^{(1),(\mathtt{SB})}, -) = f_k({q'}_0^{(1),(\mathtt{SB})}, 3).$$

If we consider all pairs (p', q') such that

$$\Delta(p'_i, q'_i) = \begin{cases} 5, \text{ if i=0} \\ 0, \text{ otherwise} \end{cases}$$

the analysis of the sbox shows that

$$Z_3 := (p'_0 \oplus k_0 = 0, q'_0 \oplus k_0 = 5)$$

and

$$Z_4 := (p'_0 \oplus k_0 = 122, q'_0 \oplus k_0 = 127)$$

are the only pairs with $\Delta_{in} = 5$ and $\Delta_{out} = 8$. Detecting one of these pairs using f_k yields again a set of 4 candidates for k_0.

Next, $\mathcal{A}$ computes the difference of plaintexts p_0 and p'_0. The difference must be one of the differences listed in Table 5.1. Since all possible differences are distinct, $\mathcal{A}$ can determine $p_0 \oplus k_0$ and hence k_0.

Cost Analysis Following the cost analysis as above this method determines the correct candidate of each key byte with 1024 faults as in the previous method plus additional 1024 faults.

	$p_0 \oplus k_0$			
$p'_0 \oplus k_0$	0	10	244	254
0	0	10	244	254
5	5	15	241	251
122	122	112	142	132
127	127	117	139	129

Table 5.1: All possible differences of p_0,p'_0

5.4.4 Fourth Attack

We assume that $\mathcal{A}$ can flip a bit of a specific byte of the intermediate state $p^{(1),(\mathtt{SB})}$. However, he has no control over the bit position. Instead, we assume that all of the 8 possible bit flips occur with the same probability 1/8. We also assume that collision information remains valid over the time span of the attack. Finally, we assume that the smartcard is not protected by a MEM.

The attack works as follows. In a first step $\mathcal{A}$ selects a set S of 256 plaintexts p that take on all different values in byte p_0 and are equal in each other byte. $\mathcal{A}$ collects the collision information $f_k(p_0^{(1),(\mathtt{SB})}, -)$ for all elements of S. Then he chooses an arbitrary plaintext q and encrypts q inducing a fault into bit e of $q_0^{(1),(\mathtt{SB})}$. By comparing the collision information $f_k(q_0^{(1),(\mathtt{SB})}, e)$ with the collision information collected in the first step $\mathcal{A}$ can determine the corresponding plaintext p_0 such that

$$\mathbf{S}[p_0 \oplus k_0] = \mathbf{S}[q_0 \oplus k_0] \oplus 2^e.$$

Note that e is unknown to $\mathcal{A}$ since he does not have any influence on the bit position. $\mathcal{A}$ can test all candidates $\widehat{k}_0$ of k_0 by simply checking if $\mathbf{S}[p_0 \oplus \widehat{k}_0] \oplus \mathbf{S}[q_0 \oplus \widehat{k}_0]$ is a power of 2. If this condition is true $\mathcal{A}$ stores $\widehat{k}_0$ as a possible key value and discards it otherwise. An analysis of the AES sbox shows that after checking all candidates a set of at most 16 candidates will remain. $\mathcal{A}$ repeats this procedure with different q_0 until only one candidate is left. Using a refined method similar to the attack in Section 5.4.1 using approximately 3 different q_0 we can determine the correct key byte with high probability. Hence, we expect that this attack needs roughly $3 \cdot 16 = 48$ faults.

5.4.5 Fifth Attack

We assume that $\mathcal{A}$ can flip a bit of a specific byte of the intermediate state $p^{(1),(\mathtt{SB})}$. However, he has no control over the bit position. Instead, we assume that all of the 8 possible bit flips in a position $b \in \{0, \ldots, 7\}$ occur with the same probability 1/8. We do not assume that collision information remains valid over the time span of the attack. Hence, $\mathcal{A}$ is only able to compare collision information of two recently obtained measurements. Finally, we assume that the smartcard is not protected by a MEM.

$\mathcal{A}$ chooses Δ_{in} of the sbox in round 1 in such a way that the number of pairs that have difference Δ_{in} and output difference with Hamming weight 1 is maximal. This choice reduces the number of faults $\mathcal{A}$ has to induce as we will see later. An analysis of the sbox shows that $\Delta_{in} = 216$ is the best choice since 8 is the maximum number of pairs that fulfill the requirements.

A single bit flip induced into $q_0^{(1)(\mathtt{SB})}$ may produce a collision if and only if $p_0 \oplus k_0$ is one of the following values:

$$0, 2, 8, 28, 29, 41, 111, 117, 173, 183, 196, 197, 208, 216, 218, 241.$$

To detect the collision f_k should have the property that

$$f_k(p_0^{(1)(\texttt{SB})}, -) = f_k(q_0^{(1)(\texttt{SB})}, b). \tag{5.7}$$

A collision implies that k_0 is an element of the set of 16 candidates

$$\begin{aligned}\mathcal{L} \;=\; & \{p_0, p_0 \oplus 2, p_0 \oplus 8, p_0 \oplus 28, p_0 \oplus 29, p_0 \oplus 41, p_0 \oplus 111, p_0 \oplus 117, p_0 \oplus 173,\\ & p_0 \oplus 183, p_0 \oplus 196, p_0 \oplus 197, p_0 \oplus 208, p_0 \oplus 216, p_0 \oplus 218, p_0 \oplus 241\}.\end{aligned}$$

To determine k_0 the adversary $\mathcal{A}$ first builds a list of all 128 pairs (p_0, q_0) of plaintexts with difference 216 in byte 0 and difference 0 in all other bytes. Then $\mathcal{A}$ selects an arbitrary q_0, derives $f_k(q_0^{(1)(\texttt{SB})}, b)$ of the corresponding plaintext and compares it with the collision information $f_k(p_0^{(1)(\texttt{SB})}, -)$ of the corresponding plaintext of p_0. $\mathcal{A}$ repeats this procedure until he detects a collision. At his point $\mathcal{A}$ knows that k_0 is an element of the set $\mathcal{L}$.

To identify the correct candidate $\mathcal{A}$ could start an exhaustive search or repeat the procedure with a different combination of input and output differences. For example $\mathcal{A}$ chooses input difference 4 and output difference 32. Since $(88, 92)$ is the only such pair $\mathcal{A}$ can use f_k as a special case of (5.7) having the property

$$f_k(p_0^{(1)(\texttt{SB})}, -) = f_k(q_0^{(1)(\texttt{SB})}, 5)$$

to test each candidate $\widehat{k}_0 \in \mathcal{L}$ of k_0.

To check whether a candidate $\widehat{k}_0 \in \mathcal{L}$ is equal to k_0, $\mathcal{A}$ derives the collision information $f_k(p_0^{(1)(\texttt{SB})}, -)$ and $f_k(q_0^{(1)(\texttt{SB})}, b)$ for $p_0 = \widehat{k}_0 \oplus 92$ and $q_0 = \widehat{k}_0 \oplus 88$. Since $(92, 88)$ is the only pair with $\Delta_{in} = 4$ and Hamming weight of $\Delta_{out} = 1$, the adversary$\mathcal{A}$ can check his hypothesis $\widehat{k}_0$. More precisely if $\widehat{k}_0 \neq k_0$ the Hamming weight of the output difference will always be greater than 1 except for the case that $p_0^{(0)(\texttt{AR})} = 88$ and $q_0^{(0)(\texttt{AR})} = 92$. But this case implies that $\widehat{k}_0 \oplus 4 = k_0$ which is impossible since every difference of two of the sixteen candidates is different from 4. So a wrong hypothesis cannot create a collision. On the other hand if $\widehat{k}_0 = k_0$ then $p \oplus k_0 = 92 \oplus \widehat{k}_0 \oplus k_0 = 92$ and $q \oplus k_0 = 88 \oplus \widehat{k}_0 \oplus k_0 = 88$ is the demanded pair and $\mathcal{A}$ will detect a collision using f_k.

Cost Analysis The success probability of finding one of the 8 pairs in part one of the attack choosing p_0 uniformly at random is $\frac{8}{128} \cdot \frac{1}{8} = \frac{1}{128}$. Hence 128 is the expected number of faults $\mathcal{A}$ has to induce.

The success probability in the second step is $(1/8) \cdot (1/16) = 1/128$. So we expect that $\mathcal{A}$ needs additional 128 faults. Hence the total number of faults to determine a key byte is $2 \cdot 128 = 256$.

To compute a complete 128 bit AES key we expect that $\mathcal{A}$ needs $16 \cdot 256 = 4096$ faults.

5.5 Conclusion

In this chapter we introduced the concept of fault based collision attacks. We showed that combining the concepts of fault attacks and collision attacks leads to powerful attacks. Fault based collision attacks do not need faulty ciphertexts but only need collision information. It turned out that this is a much weaker requirement.

Furthermore, we considered so called memory encryption mechanisms (MEM), an elaborative countermeasure widely used to protect high-end security smartcards against side channel attacks. We showed that using MEM in a straightforward manner does not increase security as much as one would expect. E.g., we presented a fault based collision attack on AES that breaks an implementation protected by a MEM by inducing only about 285 faults. Moreover, we showed how to mount further fault based collision attacks on AES in different scenarios. Table 5.2 shows an overview of the 5 attacks presented in this chapter. The first row shows the precision of the fault induction needed for each of our attacks. The second row shows whether the collision information is valid over the whole time span of the attack or if it changes after a short period of time. The third row shows if the target smartcard is protected by a MEM. The expected number of faults needed for the attack is shown in the last row.

	basic attack	attack 2	attack 3	attack 4	attack 5
Precision	high	high	high	loose	loose
coll. information valid?	yes	yes	no	yes	no
MEM	no	yes	no	no	no
# faults	32	285	1024	48	4096

Table 5.2: Overview over the fault based collision attacks

To thwart our attack one has to be more careful. Using a MEM one has to ensure that different memory encryption functions (keys) are used to protect different bytes of an intermediate state. Furthermore, we suggest to change the keys of the memory encryption frequently. Depending on the smartcard and the application one can also consider to increase the block size of the memory encryption function, e.g., to 16 bit. This would increase the complexity of fault based collision attacks.

For high-end security applications we suggest to use a randomization strategy like the one proposed in Chapter 4. Obviously, this approach is more expensive in terms of random bits. However, it provides a much better security that can be scaled to meet the desired security level.

Chapter 6

Cache Behavior Attacks (CBAs)

The performance of recent computers benefits from the progress in chip design and computer architecture. I.e., the usage of fast but small buffers, so called cache memories, improves the execution time of algorithms significantly. At first glance this helps to improve security because even more complex cryptographic algorithms, e.g., encryption algorithms could be used without slowing down the system too much. However, performance improvements often also open side channels that leak information about intermediate states of the encryption process. In this chapter we analyze and formalize the information leakage due to cache behavior.

It was first observed in (Hu 1992) and (Trostle 1998) that cache behavior opens a covert channel. They did not focus on attacking cryptographic algorithms but analyzed the multi-level security of complex systems. Later, (Kocher 1996) and (Kelsey, Schneier, Wagner and Hall 1998) were the first who mentioned that cache behavior may be a possible point of attack for cryptographic algorithms. During the selection process of AES the resistance of the candidate algorithms against side channel attacks was investigated for example in (Daemen and Rijmen 1999). At this time only the time and power consuming operations like multiplication were in the field of vision. Table lookups, e.g., for efficient application of sboxes were regarded to be resistant against side channel attacks since they were supposed to be constant time and constant power consuming. However, this turned out to be wrong.

The first theoretical *cache behavior attack* (CBA) was mounted on DES and presented in (Page 2002). Later the authors of (Tsunoo, Saito, Suzaki, Shigeri and Miyauchi 2003c) proved that cache attacks are a realistic threat for cryptographic algorithms. They performed a cache based attack on DES that successfully determined the secret key. Page extended the theoretical concept of CBAs in (Page 2003). He started to classify CBAs into time driven CBAs and trace driven CBAs depending on attackers abilities. The upcoming publications of practical attacks against AES (Bernstein 2005), (Osvik, Shamir and Tromer 2006), (Brickell, Graunke, Neve and Seifert 2006) and RSA (Percival 2005) revealed the full power of cache behavior attacks. These attacks even justify to introduce a new class of CBAs, so called access driven CBAs.

In this chapter we give the background of CBAs and present the progress in the area of CBAs up to now. After that we present a different view on how to counteract CBAs that leads to novel countermeasures. A more detailed description of the structure of this chapter is as follows:

Section 6.1: Cache Mechanism and Technical Background 73
We give a brief summary of the memory management, i.e., the cache mechanism of recent computers. All technical details that are necessary to understand CBAs and countermeasures are explained here.

Section 6.2: Security Models for CBAs 75
In this section we describe the theoretical foundations of CBAs. To analyze attacks and countermeasures one has to define the abilities of the attacker and the properties of the underlying implementation. We distinguish three different models: time driven, trace driven, and access driven CBAs. We propose to use a strengthened variation of the access driven model as a basis for security analysis and for developing countermeasures.

Section 6.3: Access Driven CBAs on AES 85
In this section we describe two concrete CBAs on AES. The first one is due to (Osvik et al. 2006). It is based on the first round(s) of AES. The second attack is more efficient and only focuses on the last round of AES. The differences of these attacks lead to a new countermeasure that we present in Section 6.7.2.

Section 6.4: General Methods to Thwart CBAs 88
This section provides a list of methods to thwart cache behavior attacks proposed so far.

Section 6.5: Information Leakage and Resistance 89
In this section we introduce the concept of information leakage and the concept of resistance to estimate the susceptibility of an implementation. Information leakage allows to estimate the uncertainty of an attacker about the secret key that remains after successfully mounting a CBA. The resistance is a measure that indicates the expected effort for an adversary to derive some information about the secret key.

Section 6.6: Information Leakage and Resistance of Selected Implementations 92
In this section we examine the information leakage and the resistance as defined in the former section of selected implementations of AES against access driven CBAs. Beside well known implementations we also consider new implementations of AES to counteract CBAs. We show that one of the new implementations is provably secure even in our strengthened access driven CBA model.

Section 6.7: Countermeasures Based on Permutations 100
The usage of random permutations is one of the countermeasures proposed in the

literature. We analyze the security a random permutation provides by describing an attack on a AES implementation protected by using a random permutation. In the sequel, we introduce so called distinguished permutations. A distinguished permutation is a permutation having a special property that ensures that some key bits are protected unconditionally. This is an improvement over the usage of general permutations that leak all bits of the secret key as our attack in Section 6.7.1 shows.

Section 6.8: Concluding Remarks 106

Finally, we recapitulate CBAs and countermeasures. We describe how to combine the proposed countermeasures to improve the ratio of security and efficiency.

6.1 Cache Mechanism and Technical Background

In this section we introduce the technical background of cache based attacks. A thorough treatment of computer architecture and memory management is given in (Hennesey and Patterson 2002). (Handy 1998) addresses the topic of cache memories even more deeply from a processor designers view.

The processor (CPU) and the main memory (RAM) are the two main building blocks of recent computers that play an important role in cache based attacks. The CPU only has very fast but few so called *CPU registers* (short: *registers*) each having the size of a processor word, e.g. 32 or 64 bits. To process data that is stored in the RAM the data has to be transferred to the CPU registers. Hence, RAM should have at least two properties:

1. RAM should be *large* in order to allow to store a lot of data.
2. RAM should be *fast* in order to allow access and process data quickly.

However, with recent technology these two properties are contradictory. Memory that has to be fast is necessarily restricted to small size and memory that has to be large is necessarily slow. In order to compensate this discrepancy, modern computers use a hierarchy of typically 4 different levels of memories that differ in size and speed. The CPU registers are placed in level 1 of the memory hierarchy. They have the shortest access time but are limited altogether to less than 1 KB. To compensate the rather slow accesses to the main memory placed in level 3 the so called *cache memory* (short: *cache*) is placed in level 2. Cache is much faster than the main memory but its size is restricted to a few megabytes. So, cache memory constitutes a trade-off between the small but fast CPU registers and the large but slow main memory[1]. The hard drive is placed in level 4 of the memory hierarchy. It is orders of magnitude larger but also orders of magnitudes slower then the other types of memories. However, the hard

[1]Note that in recent CPU's the cache memory (level 2) is split into different so called *cache level* again differing in size and speed. However, in order to simplify descriptions we stick to the simpler situation with only a single cache level.

drive has no influence on CBAs. Table 6.1 shows the memory hierarchy of recent computers. Furthermore, an overview over the typical sizes and the typical access times of memories of different levels are given.

The cache memory is divided into d so called *cache lines* each of size λ bits. The set of cache lines is partitioned into m so called *cache sets* each containing exactly d/m cache lines[2]. Likewise, the memory is divided into so called *memory blocks* of size λ bits. The memory blocks are labeled with consecutive numbers referred to as the *address* of the memory block.

Every transfer of data from the main memory is redirected through the cache. Whenever data should be transferred to the CPU it is checked whether the data is already in the cache or not. If the requested data is not in the cache the whole memory block B that contains the data is first loaded from the main memory to the cache and then the data is transferred to the processor. This is called a *cache miss*. To which cache set and cache line the data is transferred depends on the address M of the requested data. M is split into t so called *tag bits*, s so called *set bits* and b *offset bits* as depicted in Figure 6.1. The set bits determine the cache set. According to a placement strategy, one of the cache lines in the determined cache set is chosen to host the data of the memory block B. The tag bits are also stored as meta information about the content of the cache line. Since the cache is much smaller than the main memory, the previous content of the chosen cache line has to be overwritten. This is called data *eviction*.

On the other hand, if the cache already contains the requested data, it is directly transferred from the cache to the processor avoiding the access to the slow main memory. This is called a *cache hit*. Hence, if a process uses certain data more often, after the first access the data resides in the cache and can be quickly transferred to the processor. To find the requested data in the cache, the address M is split like above into set bits, tag bits and offset bits. The set bits determine the correct cache set S. The data is contained in that cache line of S whose tag bits match the tag bits of the requested data.

A cache that groups d/m cache lines in a cache set is called (d/m)*-way associative cache*.

[2] Note that d is always chosen as a multiple of m.

	register	cache	RAM	hard disc
level	1	2	3	4
typical size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
access time	$0.25 - 0.5$ ns	$0.5 - 25$ ns	$80 - 250$ ns	5 ms
hit time	-	1-2 cycles	100 cycles	10.000.000 cycles
miss penalty	-	$25 - 100$ cycles	-	-

Table 6.1: The memory hierarchy

data address		
memory block address		offset bits
← t tag bits →	← s set bits →	← b offset bits →

Figure 6.1: Partitioning the address of requested data

If $d/m = 1$ then the cache is called a *direct mapped cache*. If every memory block can be hosted by every cache line the cache is called a *full associative cache*. Figure 6.2 illustrates the different types of caches. On one hand, the larger the number d/m of cache lines per cache set is, the higher is the chance to avoid eviction of data that is still needed. On the other hand, the larger the number d/m is the longer takes it to find the requested data in the cache. Therefore, most recent processors use direct mapped caches. Some processors use 2- or 4-way associative caches.

6.2 Security Models for CBAs

In this section we present general principles of how to exploit the knowledge about the cache behavior to determine information about intermediate results of an algorithm. In turn, this information can be used to derive information about the secret key of a cryptographic

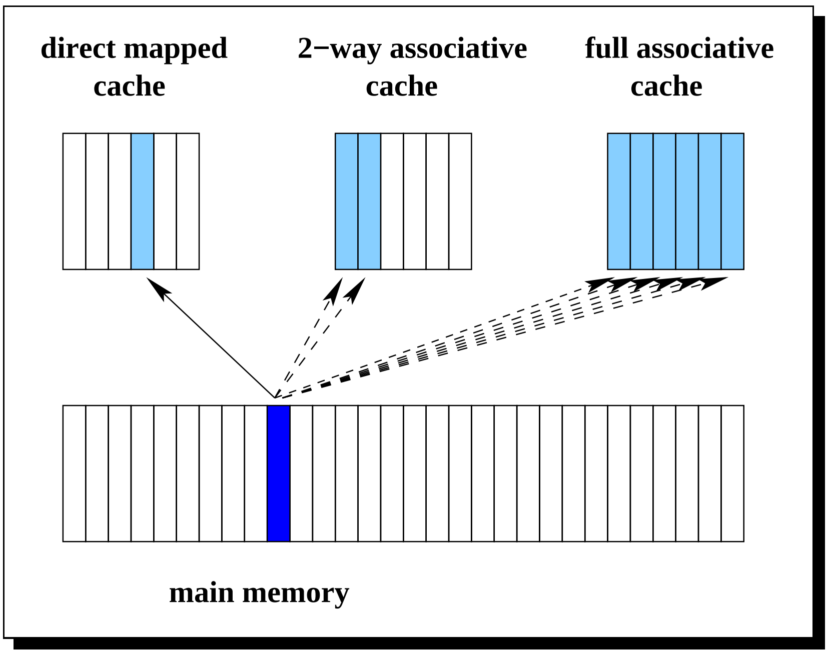

Figure 6.2: Different types of cache memory

algorithm. In the next section we describe the basic setting and the basic abilities of the adversary referred to as the *fundamental model*. After that we present three different threat models for CBAs that are based on the fundamental model.

6.2.1 Fundamental Model for CBAs

We consider a so called *crypto process* running on a computer with cache memory. This crypto process encrypts (or decrypts) a given plaintext (or ciphertext) using a secret key k. The adversary $\mathcal{A}$ wants to derive information about k by analyzing plaintext/ciphertext pairs. Depending on the underlying threat model $\mathcal{A}$ gets some additional side channel information that leaks due to the cache mechanism. I.e., we focus on the security problems based on sharing the cache between processes. However, reading data of other processes directly is prevented by the memory management. The only interaction that happens is the mutual eviction of data.

To be more specific, in the fundamental threat model for CBAs we assume that the following holds:

Assumption 11 *$\mathcal{A}$ knows all technical details about the underlying cryptographic algorithm and its implementation (*Kerckhoffs' extended principle*).*

Assumption 12 *Every memory block of the sboxes is mapped to a different cache line. I.e., the applications of the sboxes do not cause any data eviction of sbox data.*

Assumption 13 *During the attack only cache accesses caused by the encryption occur.*

Assumption 14 *In the beginning of an encryption / decryption no sbox data is stored in the cache.*

Assumption 15 *$\mathcal{A}$ can feed the crypto process with known (or chosen) plaintexts (or ciphertexts) and obtains the corresponding ciphertexts (or plaintexts).*

Discussion of the Fundamental Model In the following we discuss and justify the fundamental model for CBAs as given above. Variations of how to implement a cryptographic algorithm efficiently are rather limited. Hence, the security of an implementation should not rely on keeping implementational aspects secret. We call this *Kerckhoffs' extended principle* according to Kerckhoffs' principle (Kerckhoffs 1883).

In this thesis we focus on the symmetric cipher AES [3]. Beside the standard implementation we consider several variations of the fast implementation of AES as described in Section 2.4 (page 16). This implementation uses 5 sboxes $\mathbf{T}_0, \ldots, \mathbf{T}_4$ each having 256 entries of size 4 bytes.

Since we focus on information leakage due to table lookups, we assume that $\mathcal{A}$ knows the position of these sboxes in the memory and the possible cache lines they can be mapped to. Recent processors possess several megabytes of cache memory that can hold the much smaller sbox data completely. Hence, an access to some sbox data cannot evict other sbox data. To simplify the description further, we assume that each sbox is mapped consecutively into the cache memory. Let v be the number of sbox elements that can be stored in a single cache line. For each sbox $\mathbf{T}_j$, $0 \leq j \leq 4$ and $0 \leq i \leq \lceil \frac{256}{v} \rceil - 1$ let CL_i^j denote the cache line that contains the following elements:

$$CL_i^j = [\mathbf{T}_j[i \cdot v], \ldots, \mathbf{T}_j[i \cdot v + v - 1]] \,.$$

If it is clear from the context which is the referred sbox we simply write

$$CL_i = [\mathbf{S}[i \cdot v], \ldots, \mathbf{S}[i \cdot v + v - 1]] \,.$$

Furthermore, for an index x of an sbox entry let $\langle\ x\ \rangle$ denote the index of the cache line that stores x. For example $\langle\ \mathbf{T}_j[i \cdot v + 1]\ \rangle = i$. Remember that $\mathcal{A}$ knows all technical details about the implementation in particular the position and the mapping of the sboxes to the cache lines. Hence, $\mathcal{A}$ can compute $\langle\ x\ \rangle$ for every x efficiently.

State of the art encryption algorithms like AES are very fast. Therefore, it is very unlikely that the encryption of a single plaintext is interrupted by an other process accidentally. This implies that during the encryption no other process causes cache accesses. Additionally, we assume that in the beginning of every encryption (decryption) the cache does not hold any data of an sbox. Hence, cache hits and cache misses only depend on the actual encryption process. In particular, former encryptions do not have any influence on the cache content.

Note that the Assumptions 12 through 14 of the fundamental CBA model above improve the strength of an adversary. They allow a simpler analysis and a simpler description of CBAs but are not essential for an attack to be successful in principle. However, if the Assumptions 12 through 14 do not hold the complexity of an attack increases.

In (Page 2003) two general approaches are given to classify models of cache behavior attacks: the trace driven CBA and the time driven CBA.

6.2.2 Time Driven CBA

As described in Section 3.2.1 it is possible to use timings of encryptions to determine information about the secret key. The classical timing attacks on RSA (Kocher 1996, Dhem

[3]However, all the analysis can be adapted to implementations of virtually any block cipher that uses table lookups.

et al. 1998) and AES (Koeune and Quisquater 1999) are based on data dependend timings of certain operations during the encryption, e.g., multiplication. However, in modern block ciphers like DES and AES, a complex function like the non-linear substitution is usually realized via table lookups. In the AES selection phase it was not clear how to mount side channel attacks based on table lookups. Table lookups were regarded as constant time operations and therefore regarded as resistant against timing attacks, see (Daemen and Rijmen 1999).

As it turned out, this is not true for implementations running on computers with cache. On computers with cache, table lookups to some indices will cause a cache hit while table lookups to other indices cause a cache miss. An element of an sbox that is already stored in the cache can be accessed faster than an element that is not stored in the cache. The index of such an sbox lookup depends on values of intermediate results that again depend on the plaintext and the secret key.

Hence, values of intermediate results indirectly influence the running time of the algorithm, even for table lookups. These data dependend timings can be statistically analyzed by an attacker $\mathcal{A}$ to derive information about intermediate states. In turn, information about intermediate states let $\mathcal{A}$ deduce information about the secret key k. Hence, there is information leakage due to the cache behavior of the cryptographic algorithm.

Threat Model for Time Driven CBAs The threat model for time driven CBAs is based on the fundamental threat model presented in Section 6.2.1 (page 76). For a time driven CBA to be successful, the following assumptions must be valid:

Assumption 18 *It is more likely that an encryption of a plaintext that causes only few cache misses has a short running time than an encryption of a plaintext that causes more cache misses.*

Assumption 19 $\mathcal{A}$ *is able to measure the time an encryption takes with reasonable precision.*

Discussion of the Threat Model Assumption 18 specifies the relation between the cache behavior and the overall encryption time. The impact of a single cache hit or miss on the encryption time depends on the underlying hardware. To mount a time driven CBA we assume (Assumption 19) that the attacker can measure the encryption time. The precision of these measurements is sufficient to allow a statistical analysis of the timings to verify if a cache hit or miss occurred during a certain step of the encryption. In general, an attacker $\mathcal{A}$ does not need complex equipment or techniques to measure the running time with reasonable precision. For example, modern processors provide so called performance registers, e.g., to measure timings of processes with a resolution in the range of clock cycles. A description of how to use the performance registers, e.g., for time measurements is given in (Intel 2006).

Basic Structure of Time Driven CBAs The basic structure of a time driven CBA follows the structure of general side channel attacks. In order to determine information about the i-th byte k_i of the secret key k the attack consists of the two steps shown in Figure 6.3.

measurement step: An attacker $\mathcal{A}$ chooses a set S of $n \in \mathbb{N}$ arbitrary but different plaintexts $p^{(1)}, \ldots, p^{(n)}$. For each of these plaintexts $p^{(j)}$, $\mathcal{A}$ measures the time $t^{(j)}$ the crypto process needs to encrypt $p^{(j)}$.

analysis step: To test a hypothesis $\widehat{k}_i$ of the i-th byte k_i of the secret key k the attacker $\mathcal{A}$ uses the following method based on the method described in (Dhem et al. 1998).

1. $\mathcal{A}$ reproduces a part of the encryption of $p^{(j)}$ assuming that $\widehat{k}_i$ is correct. In particular, $\mathcal{A}$ computes a certain intermediate result $\widehat{x}^{(j)}$ of the encryption of $p^{(j)}$ that only depends on the plaintext, the candidate $\widehat{k}_i$ and possibly on other parts of the key that $\mathcal{A}$ already knows. E.g., in AES this could be a byte of the state after the first application of the sbox.
2. Furthermore, $\mathcal{A}$ simulates the cache behavior of the encryption on that computer. Hence, $\mathcal{A}$ determines the number $z^{(j)}$ of cache misses that occur during the computation of $\widehat{x}^{(j)}$. Let
$$M = \frac{1}{n} \cdot \sum_{j=1}^{n} z^{(j)}$$
denote the average number of cache misses taken over all $z^{(j)}$.
3. $\mathcal{A}$ partitions the set S of plaintexts into two sets S_s and S_l as follows. A plaintext $p^{(j)}$ is placed in set S_s if the number of cache misses that occur during the computation of $\widehat{x}^{(j)}$ is less than M. Otherwise $p^{(j)}$ is placed in set S_l.
4. $\mathcal{A}$ computes the mean encryption times M_s and M_l of plaintexts in S_s and S_l as
$$M_s = \frac{1}{n} \sum_{p^{(j)} \in S_s} t^{(j)}$$
and
$$M_l = \frac{1}{n} \sum_{p^{(j)} \in S_l} t^{(j)}.$$
If M_s and M_l differ significantly $\mathcal{A}$ concludes that the candidate $\widehat{k}_i$ is correct. In the other case, $\mathcal{A}$ concludes that the candidate is wrong.

Figure 6.3: Basic structure of a time driven CBA

To see why the attack works we first consider the case that the candidate $\widehat{k}_i$ is correct. Hence, it is more likely that $z^{(j)}$ matches the number of cache misses that occur during the computation of $x^{(j)}$ in the encryption. Due to Assumption 18, it is more likely that an encryption of a plaintext $p^{(j)}$ that causes only few cache misses while computing $x^{(j)}$ has a shorter running time than an encryption that causes many cache misses. Therefore, M_l should be significantly larger than M_s.

If $\widehat{k}_i$ is not correct it is likely that the $z^{(j)}$ are not the correct numbers of cache misses that occur during the computation of $x^{(j)}$. Hence, the partition of the plaintexts into the sets S_s and S_l is not entirely determined by the correct number of cache misses. We expect that the mean times M_s and M_l of both sets do not differ significantly.

The success probability of the attack depends on the precision of time measurements and on the number n of plaintexts. Improving the precision of the measurements and increasing the number n of plaintexts increases the success probability.

The first time driven CBAs mounted on DES, AES and several other block ciphers where published in (Tsunoo, Tsujihara, Minematsu and Miyauchi 2002), (Tsunoo, Kubo, Shigeri, Tsujihara and Miyauchi 2003a), (Tsunoo, Kawabata, Tsujihara, Minematsu and Miyauchi 2003b), (Tsunoo et al. 2003c), (Tsunoo, Suzaki, Saito, Kawabata and Miyauchi 2003d) and (Tsunoo, Tsujihara, Shigeri, Kubo and Minematsu 2006).

6.2.3 Trace Driven CBA

In a trace driven CBA the attacker $\mathcal{A}$ is more powerful. We assume that $\mathcal{A}$ is able to derive the profile of the cache behavior. That means that for each memory access $\mathcal{A}$ gets the information if a cache hit or a cache miss occurred. Furthermore, it is assumed that $\mathcal{A}$ is able to relate this information to operations of the encryption. The sequence of operations together with the information whether a cache hit or miss occurred is called a *cache trace*. In the sequel, we present a threat model for trace driven CBAs to formalize the abilities of the attacker.

Threat Model for Trace Driven CBAs As for time driven CBAs the threat model for trace driven CBAs is based on the fundamental model of Section 6.2.1 (page 76). The fundamental threat model is extended by the ability of the adversary to obtain cache traces of an encryption.

Assumption 21 *$\mathcal{A}$ is able to obtain the trace of cache activity.*

In order to get a simpler description of the basic structure of trace driven CBAs we assume that $\mathcal{A}$ always gets the correct trace without any distortion. This simplification reduces the complexity but is not essential for a trace driven CBA to be successful.

Discussion of the Threat Model Assumption 21 provides the basis of trace driven CBAs. However, obtaining traces of encryptions is not as easy as simple time measurements. The attacker needs more sophisticated tools to mount a trace driven CBA. For example, Page (Page 2002) proposes power analysis or the analysis of electromagnetic radiation as means to determine cache traces. In (Bertoni, Zaccaria, Breveglieri, Monchiero and Palermo 2005) the authors show how to obtain cache traces via power analysis.

Basic Structure of Trace Driven Attacks As for time driven CBAs, the basic structure of trace driven CBAs follows the structure of general side channel attacks. In order to determine information about the i-th byte k_i of the secret key k the attack consists of 2 steps shown in Figure 6.4.

measurement step: $\mathcal{A}$ chooses a set S of $n \in \mathbb{N}$ plaintexts $p^{(1)}, \ldots, p^{(n)}$ and obtains the cache trace of the encryption of each $p^{(j)}$ as explained above.

analysis step: To test a hypothesis $\widehat{k_i}$ of a byte k_i of the secret key k the adversary uses the following method.

1. $\mathcal{A}$ reproduces a part of the encryption of $p^{(j)}$ assuming that $\widehat{k_i}$ is correct. In particular, $\mathcal{A}$ computes a certain intermediate result $\widehat{x}^{(j)}$ of the encryption of $p^{(j)}$ that only depends on the plaintext, the key byte $\widehat{k_i}$ and possibly on other parts of the key that $\mathcal{A}$ already knows. E.g., in AES this could be a byte of the state after the first application of the sbox.
2. Furthermore, $\mathcal{A}$ simulates the cache behavior of the encryption on that computer. Hence, $\mathcal{A}$ determines the cache trace that occurs during the computation of $\widehat{x}^{(j)}$. If the trace of the simulated cache behavior that occurs during the computation of $\widehat{x}^{(j)}$ matches the obtained cache trace the hypothesis may be correct. Otherwise the hypothesis is proven to be wrong.

Figure 6.4: Basic structure of a trace driven CBA

Examples for trace driven CBAs are given in (Page 2002) and (Acıiçmez and Koç 2006).

6.2.4 Access Driven CBA

In this section we present a threat model that is stronger than the models presented above. In addition to the plaintext/ciphertext pair, the adversary $\mathcal{A}$ gets the information which cache lines were accessed during the encryption. Strengthening the threat model in this way is justified by the attacks of (Bernstein 2005), (Osvik et al. 2006) and (Neve and Seifert 2006). These attacks show that cache based attacks are indeed very powerful, even in practice. Hence, a conservative attitude towards unclear aspects of $\mathcal{A}$'s technical abilities is necessary to get a reliable analysis.

Threat Model for Access Driven Attacks According to the models described so far access driven CBAs are also based on the fundamental threat model of Section 6.2.1 (page 76). We call this threat model the *ad CBA model*. We extend the fundamental model by assuming that the following holds:

Assumption 24 *$\mathcal{A}$ gets the indices of the cache lines that were accessed during the encryption (decryption). We call this information* cache information*.*

Assumption 25 *We explicitly assume that $\mathcal{A}$ cannot distinguish between elements in a single cache line.*

The main point is that the adversary $\mathcal{A}$ is able to determine information about which cache lines were accessed during the encryption of a plaintext. To build a strong model we simplify the determination of accessed cache lines in the following way. We assume that $\mathcal{A}$ simply gets the *correct* partition of the set of all cache lines D into the sets of indices of accessed cache lines D_0 and the set D_1 of indices of cache lines that were not accessed during the encryption of the plaintext p into the ciphertext c. We call this partition *cache information*. The triple (p, D_0, D_1) (or (c, D_0, D_1)) is called a *measurement*.

Discussion of the Threat Model Assumption 24 provides the basis of access driven CBAs. In (Hu 1992) the author already presented a method to determine the indices of cache lines that were accessed during a computation. Assuming that $\mathcal{A}$ has access to the computer he can measure the time it takes to access certain data with reasonable precision. Contrarily to the time driven CBA, $\mathcal{A}$ does not need to measure timings of the encryption process. He only needs to measure the time it takes to access parts of his own data. See (Intel 1997) for a description of how to do precise time measurements on a PC. To detect which cache lines has been accessed during the encryption $\mathcal{A}$ can use the Prime-and-Probe method shown in Figure 6.5.

If, on one hand, the crypto process accesses the cache line CL_i during the encryption he evicts the data block B_i from the cache. Hence, accessing B_i after the encryption causes a

1. Flush the cache by accessing d memory blocks $B_1, \ldots, B_d$ such that B_i is mapped to cache line CL_i.
2. Trigger the crypto process to encrypt the plaintext p.
3. For each memory block B_i , $1 \leq i \leq d$ do
   (a) measure time t to access B_i
   (b) if t is large then cache line CL_i has been accessed during the encryption
   (c) else cache line i has not been accessed by the encryption

Figure 6.5: Prime-and-Probe method

cache miss which in turn results in a larger access time. On the other hand, if the crypto process does not access cache line CL_i the data block B_i remains in the cache. Hence, accessing B_i after the encryption of p causes a cache hit, allowing to access B_i very fast.

However, we assume that $\mathcal{A}$ cannot distinguish elements of a single cache line. Up to now it is not clear if it is technically possible to distinguish accesses to elements within the same cache line. No access driven CBA published so far requires this somewhat difficult and unlikely ability of the adversary $\mathcal{A}$. Obviously, the ability to distinguish elements within the same cache line would allow even more powerful cache attacks than the attacks published so far. As we will see, all efficient countermeasure are implicitly based on this assumption.

Basic Structure of Access Driven Attacks Next we give the general structure of an access driven CBA to show how an attacker $\mathcal{A}$ can use cache information to derive information about the secret key. The attacker $\mathcal{A}$ performs the two steps shown in Figure 6.6.

At this point $\mathcal{A}$ has computed a set $\widehat{K}_i$ of possible key candidates for k_i. He knows that one of the elements of $\widehat{K}_i$ is the correct key byte k_i because $k_i \in \widehat{K}_i^{(j)}$ for all $1 \leq j \leq n$. Hence, the correct value is also an element of the intersection of all sets $\widehat{K}_i^{(j)}$.

Wrong key candidates occur for two reasons. Firstly, each access to a cache line does not determine the intermediate result exactly but leaves v possible values where v is the number of sbox elements that are stored in a single cache line. Secondly, there occur sbox lookups during the encryption that do not compute $x^{(j)}$ directly but also induce cache accesses. We call these sbox lookups *perturbing lookups*. Since an adversary cannot decide whether an sbox lookup is perturbing or not he has to consider all key candidates that cause an access to a cache line of the set D_0.

The number of the remaining candidates depends on the number of measurements and, as we will see later, on specific details of the attack. We present an access driven CBA that

measurement step: $\mathcal{A}$ gets $n \in \mathbb{N}$ measurements $m^{(1)}, \dots, m^{(n)}$ of encryptions of plaintexts $p^{(1)}, \dots, p^{(n)}$ with the secret key k. That means, for each plaintext $p^{(1)}, \dots, p^{(n)}$ the adversary $\mathcal{A}$ knows the partition of the set of all cache lines into the set D_0 of accessed cache lines and into the set D_1 of cache lines that were not accessed during the encryption.

analysis step: For each measurement $m^{(j)}$ the attacker $\mathcal{A}$ analyses the corresponding cache information to compute a set of possible values of an intermediate result $x_i^{(j)}$ of the encryption of $p^{(j)}$ that only depends on the plaintext (or ciphertext) and on the i-th byte k_i of the (round-)key k. Then $\mathcal{A}$ computes a set $\widehat{K}_i^{(j)}$ of candidates for k_i that would produce one of the possible values for $x_i^{(j)}$ during the encryption. Finally, $\mathcal{A}$ combines the information of all measurements $m^{(1)}, \dots, m^{(n)}$ by computing

$$\widehat{K}_i := \bigcap_{j=1}^{n} \widehat{K}_i^{(j)}.$$

Figure 6.6: Formal outline of an access driven CBA

can only determine half of the key bits whereas another attack that we present reveals the complete key.

6.2.5 Extending the Threat Model for Access Driven CBAs

We present an extended threat model that strengthens the attack compared to the adversary of the access driven CBA threat model. We call this model *ead CBA model*. In addition to the assumptions of access driven CBAs as described above the following assumption holds:

Assumption 27 *$\mathcal{A}$ can restrict cache information to certain rounds of the encryption.*

We assume that the adversary can influence the start and end of a measurement. I.e., $\mathcal{A}$ can restrict cache information to certain rounds of the encryption. Hence, $\mathcal{A}$ can focus on chosen rounds of the AES encryption (decryption). As we will see, restricting the cache information to certain rounds decreases the expected number of accessed cache lines. In turn this improves the complexity of access driven CBAs significantly but does not increase the information that leaks through the cache behavior of the crypto process.

Restricting measurements to certain rounds is justified by the property of modern multitasking operating systems to change the active process after a constant amount of running time. For example, see (Stallings 2005) for further details. Hence, it is possible that the encryption process is interrupted by the attackers process, allowing $\mathcal{A}$ to access the cache during an encryption (decryption). In (Bernstein 2005) Bernstein already warned that this property may be exploitable and the authors of (Brickell et al. 2006) managed to exploit it to determine cache information of arbitrary rounds on a real PC with some reasonable precision. Later, we will use the ead CBA model to analyze the resistance of implementations and countermeasures against CBAs.

Table 6.2 compares the three different types of CBAs described above. The first column indicates how difficult it is to mount the attack. The second column lists how many measurements have to be done. In the next section we give the descriptions of two access driven CBAs on AES based on the first round(s) and on the last round.

6.3 Access Driven CBAs on AES

To illustrate the general structure of access driven CBAs in the ead CBA modell, in this section we present two access driven CBAs on AES. The first attack as presented in (Osvik et al. 2006) is based on the first round(s) of AES. The second attack is based on the last round of AES. The idea was mentioned in (Osvik et al. 2006) and (Brickell et al. 2006). In the sequel we describe both attacks on the fast implementation of AES (see Section 2.4). Although both attacks work for different sizes of cache lines, we simplify the descriptions by fixing the size of a cache line to $\lambda = 512$ bits. Hence, each cache line can store $v = 16$ entries of a large sbox $\mathbf{T}_0, \ldots, \mathbf{T}_4$ and each sbox $\mathbf{T}_j$ fits into $m = 16$ cache lines $CL_0^j, \ldots, CL_{15}^j$. For $0 \leq \ell \leq 15$ the sbox the attack focus on is mapped into the cache lines as follows:

$$C_\ell^j = \{\mathbf{T}_j[x] \mid x = \ell \cdot 16, \ldots, \ell \cdot 16 + 15\}.$$

6.3.1 Access Driven CBA on the First Round

The first CBA is based on intermediate results of the first round. To be more precise, $\mathcal{A}$ focus on the result of the first application of an sbox in the first round. Since the involved

type	difficulty	complexity
time driven	low	high
trace driven	high	medium
access driven	low	low

Table 6.2: Comparing properties of different CBAs

sbox depends on the index i of the key byte we only consider the output

$$x_i = \mathbf{T}_{(i \bmod 4)}[p_i \oplus k_i]$$

of the sbox $\mathbf{T}_{(i \mod 4)}$. To simplify notation we simply write

$$x_i = \mathbf{T}[p_i \oplus k_i].$$

Structure of the Attack

To derive information about the i-th byte k_i of the secret key k the attacker performs the following operations according to the basic structure of access driven CBAs shown in Section 6.2.4 (page 83):

1. $\mathcal{A}$ chooses $n \in \mathbb{N}$ plaintexts $p^{(1)}, \ldots, p^{(n)}$ that are fixed in byte $p_i^{(j)}$ and are independent and uniformly distributed in the other bytes.
2. $\mathcal{A}$ obtains measurements $m^{(j)} = (D_0^{(j)}, D_1^{(j)}, p^{(j)})$ for $1 \leq j \leq n$.
3. $\mathcal{A}$ concludes that
$$x \in \widehat{X}^{(j)} := \bigcup_{\ell \in D_0^{(j)}} \{\ell \cdot 16, \ldots, \ell \cdot 16 + 15\}$$
4. $\mathcal{A}$ computes the sets
$$\widehat{K}_i^{(j)} = \left\{ p_i^{(j)} \oplus \widehat{x}_i^{(j)} \mid \widehat{x}_i^{(j)} \in \widehat{X}^{(j)} \right\}$$
for all $1 \leq j \leq n$.
5. $\mathcal{A}$ computes the set
$$\widehat{K}_i = \bigcap_{j=1}^{n} \widehat{K}_i^{(j)}$$
of candidates for k_i.

Discussion of the Attack Let us assume that $\mathcal{A}$ can restrict the measurements to the first round. $D_0^{(j)}$ is the set of the indices of the 16 cache lines that were accessed during the 4 applications of $\mathbf{T}_{(i \mod 4)}$ in round 1 of the encryption of the plaintext $p^{(j)}$. Hence, the correct key byte k_i is an element of every $\widehat{K}_i^{(j)}$. Remember that a cache line can store $v = 16$ elements of an sbox. Hence, depending on the plaintexts $p^{(1)}, \ldots, p^{(n)}$ the remaining set of key candidates $\widehat{K}_i$ contains at most $4 \cdot 16 = 64$ elements if all n measurements cause the access of the same 4 cache lines. However, fixing byte p_i of the plaintexts and choosing all other bytes uniformly at random lets $\mathcal{A}$ determine the cache line ℓ that is accessed while computing x after only few measurements. Knowing ℓ lets $\mathcal{A}$ reduce the number of possible key candidates to 16. To see why at least 16 key candidates will survive this attack we look at the structure of the elements of a set $\widehat{K}_i^{(j)}$. The elements of $\widehat{K}_i^{(j)}$ are always of the form

$p_i^{(j)} \oplus \ell, \ldots, p_i^{(j)} \oplus (\ell + 15)$. That means that the elements of each $\widehat{K}_i^{(j)}$ restricted to the 4 lower bits take on all 16 possible values. It follows that the attack is not able to determine the lower 4 bits of the key byte and hence $2^4 = 16$ candidates for k_i remain. In the case that $\mathcal{A}$ cannot restrict the cache information to the first round the set D_0 also contains indices of the perturbing lookups of cache lines that were accessed in rounds 2 to round 9. Hence, it will take more measurements to determine information about the key. The total amount of information that $\mathcal{A}$ gets are again the upper 4 bits of each key byte.

To determine the remaining bits of each key byte one can combine this attack with a modified attack on the second round to compute the complete key.

6.3.2 Access Driven CBA on the Last Round

In this section we describe a CBA that is based on an intermediate result[4]

$$x_i = \mathbf{S}^{-1}\left[c_i \oplus k_i^{10}\right]$$

where $\mathcal{A}$ uses cache information about the sbox lookup of the last round to determine the secret key k.

Basing the attack on the last round has advantages over the attack on the first rounds. First, cache information of the last round is sufficient to determine all bits of the secret key. So $\mathcal{A}$ does not need to attack different rounds. Another advantage is that the sbox $\mathbf{T}_4$ of the last round is special and is only used in that round. This helps the attacker because cache information is never perturbed by cache accesses of other rounds. The cache information is restricted to the last round automatically.

For sake of simplicity, we only show how to compute a single byte k_i^{10} of the last round key k^{10}. However, the same strategy can by applied to determine the other key bytes of k^{10}. Knowing all key bytes of the last round key allows to revert the key schedule and compute the cipher key k. As mentioned above, we fix the size of a cache line to $\lambda = 512$ bits and only consider the sbox $\mathbf{T}_4$ of the fast implementation of AES as described in Section 2.4 (page 16) since it is widely used in common crypto libraries like openssl (OpenSSL Project 2005). We denote the j-th cache line used for the table lookups for $\mathbf{T}_4$ by $CL_j, j = 0, \ldots, 15$. Hence, CL_j contains the 4-tuples

$$\{(\mathbf{S}[x], \mathbf{S}[x], \mathbf{S}[x], \mathbf{S}[x]) \mid x = 16 \cdot j, \ldots, 16 \cdot j + 15\}$$

as defined in Section 2.4 (page 16).

Structure of the Attack The structure of the attack on the 10th round is similar to the structure of the attack on the first round. To derive information about the i-th byte of the last round key k^{10} the attacker performs the following operations:

[4]To simplify notation we omitted the `ShiftRows` operation.

1. $\mathcal{A}$ chooses $n \in \mathbb{N}$ plaintexts $p^{(1)}, \ldots, p^{(n)}$ uniformly at random.

2. $\mathcal{A}$ obtains the ciphertexts and the measurements $m^{(j)} = (D_0^{(j)}, D_1^{(j)}, c^{(j)})$ for $1 \leq j \leq n$.

3. $\mathcal{A}$ concludes that
$$x_i^{(j)} \in \widehat{X}_i^{(j)} := \bigcup_{\ell \in D_0^{(j)}} \{\ell \cdot 16, \ldots, \ell \cdot 16 + 15\}$$

4. $\mathcal{A}$ computes the sets
$$\widehat{K}_i^{(j)} = \left\{ c_i^{(j)} \oplus \mathbf{S}\left[\widehat{x}_i^{(j)}\right] \mid \widehat{x}_i^{(j)} \in \widehat{X}_i^{(j)} \right\}$$
for all $1 \leq j \leq n$.

5. $\mathcal{A}$ computes the set
$$\widehat{K}_i = \bigcap_{j=1}^{n} \widehat{K}_i^{(j)}$$
of candidates for k_i^{10}.

If $\widehat{K}_i$ contains only a single element, the adversary has determined k_i^{10}. Now it is not hard to see that the intersection of sets in step 5 eventually will contain only a single element if every wrong key candidate is not an element of all sets $\widehat{K}_i^{(j)}$. The big difference between the attack on the first and on the last round is that in step 4 the sbox is involved in computing the intermediate result. We verified that unlike in the attack on the first round the diffusion on the bits caused by the sbox lets $\mathcal{A}$ detect wrong key candidates. That means that for every wrong key candidate $\widehat{k}$ there exist appropriate choices of plaintexts such that the resulting set of key candidates does not contain the wrong candidate $\widehat{k}$. We will consider this property of the attack more closely in Section 6.7. Moreover, experiments show that on average approximately 15 pairs $(p^{(j)}, c^{(j)})$ together with the cache information $D_0^{(j)}$ suffice to determine the key byte k_i^{10} uniquely.

6.4 General Methods to Thwart CBAs

In this section we give an overview over countermeasures to hedge implementations of cryptographic algorithms against CBAs as proposed for example in (Page 2003). We give a brief description and assess each countermeasure with respect to performance and security.

remove cache A straightforward countermeasure to counteract CBAs to remove or disable the cache and hence the cache effects. On one hand it is not clear how to do this on recent processors. On the other hand, disabling the cache would have devastating consequences on the performance of implementations.

minimize time accuracy Time driven CBAs depend on the ability of the attacker to measure timings with reasonable precision. Disturbing timing measurements, e.g., by inserting random dummy operations into the encryption process, would increase the effort for an attacker. However, a time driven CBA may still be feasible.

maximize line size The size of a cache line determines the amount of information that leaks by a CBA. The larger a cache line is, the lower is the amount of information that leaks. This also increases the effort needed to perform a CBA but does not necessarily prevent it.

perform cache warming Warming the cache, that means loading the whole sbox into the cache before starting the encryption, was first regarded as an effective countermeasure. However, Bernstein (Bernstein 2005) warned about the effectiveness and the authors of (Brickell et al. 2006) managed to defeat this countermeasure.

disable cache flushing Another point of defense could be to prevent an attacker from flushing the cache. In combination with cache warming this would render all CBAs useless. However, building this countermeasure needs additional hardware support that would be very expensive.

cache flushing on every process switch In the analysis of the VAX security kernel (Hu 1992) the author proposes to clear the cache on every process switch. This approach needs the support of the kernel of the operating system and would obviously close the cache based side channel. However, even with hardware acceleration the impact on the performance would be very high.

randomize the instruction order Randomizing the instruction order could also increase the effort to mount CBAs. Because the attacker cannot associate side channel information to certain operations, the number of measurements needed to deduce information about intermediate states increases. See (May, Muller and Smart 2001a) and (May, Muller and Smart 2001b).

randomize intermediate states Randomizing intermediate states as described in Chapter 4 obviously thwarts CBAs. Each intermediate result is completely randomized such that is independent of the plaintext and the secret key. Hence, even if table lookups are used to compute intermediate results the information that leaks via a CBA is also independent of the secret key.

6.5 Information Leakage and Resistance

CBAs are very powerful attacks. Although they seem to be unrealistic and hypothetical on first sight they were proven to be a real threat for implementations of cryptographic algorithms on computers with cache. Hence, a strong threat model is essential for a thorough security analysis. The threat model described above is stronger than the threat models published so far. The adversary is more powerful because $\mathcal{A}$ can restrict the cache information to a smaller interval of encryption operations. This reduces the number of accessed cache lines per measurement and increases the efficiency of cache based attacks. The main questions when analysing the security against CBAs are information leakage and complexity of a CBA. After giving a formal definition of information leakage we introduce the notion of the so called *resistance* of an implementation as a measure that allows to estimate the complexity of a CBA.

Information Leakage The most important aspect of an implementation regarding the security against access driven CBAs is to determine the maximal amount of information that leaks via access driven CBAs. As we will see, the amount of leaking information about the secret key varies depending on the details of the CBA and the implementation of the cryptographic algorithm. We make the following definition:

Definition 3 (information leakage) *We consider an adversary who can mount a CBA using an arbitrary number of measurements. Let $\widehat{\mathcal{K}}_i$ be the set of remaining key candidates for a key byte k_i^{10} at the end of the attack. Then the leaking information is*

$$8 - \log_2\left(|\widehat{\mathcal{K}}_i|\right)$$

bits.

The amount of leaking information allows to estimate the uncertainty of an attacker about the secret key that remains after a successful access driven CBA. To quantify the maximal amount of information $\mathcal{A}$ can obtain about the secret key by access driven CBAs, we define $|CL|$ to be the size of a cache line in bits, $|S|$ the number of entries of the sbox and s the size of a single sbox element in bits. Hence, the number of elements that fits into a cache line is $\frac{|CL|}{s}$ and the cache information of a single measurement leaks at most

$$\log_2(|S|) - \log_2\left(\frac{|CL|}{s}\right) = \log_2\left(\frac{|S|}{|CL|} \cdot s\right)$$

bits. Depending on the exact nature of an attack, the sets of measurements let the attacker reduce the number of remaining key candidates after the attack. The information leakage varies between 0 and 8 bits of information per byte. For example, the attack on the first round of (Osvik et al. 2006) mounted on the fast implementation can determine at most 4 bits of every key byte regardless of the number of measurements. In contrast, the attack of (Brickell et al. 2006) based on the last round allows an adversary to determine all key bits. Furthermore, in Section 6.6 (page 96) we present an implementation that does not leak any information in our model.

Complexity of a CBA The information leakage as defined above measures the maximal amount of information a CBA can provide using an arbitrary number of measurements. Determining the expected number of measurements an attacker needs to obtain the complete leaking information depends on the details of the implementation and on details of the CBA. For simplification we introduce the notion of so called resistance. The resistance focuses on the general structure of a CBA as shown in Section 6.2.4 (page 83) and does not consider details of certain CBAs. It is a general measure to estimate the complexity of CBAs on different implementations.

Definition 4 (Resistance) *The resistance of an implementation is the expected number E_r of key candidates that are proven to be wrong during a single measurement that is based on r rounds of the encryption.*

The larger E_r the more susceptible is the implementation to access driven CBAs. In particular, if an implementation does not leak any information then an adversary cannot rule out key candidates and hence the resistance is 0. To compute E_r we assume that all sbox lookups are independently and uniformly distributed. This assumption is justified because an attacker $\mathcal{A}$ usually does not have any information about the distribution of the sbox lookups. Hence, the best he can do in an attack is to choose the parts of the plaintexts/ciphertexts that are not relevant for the attack uniformly at random.

Let m be the number of cache lines needed to store the complete sbox. Each cache line can store v elements of an sbox. Furthermore, let w be the number of sbox lookups per round and let r be the number of rounds the attack focuses on. In an access driven CBA a key candidate is proven to be incorrect if it causes an access of a cache line that was not accessed during a measurement. Assuming that all sbox lookups are uniformly distributed the probability that a cache line is not accessed in all $r \cdot w$ sbox lookups is

$$p_{\text{miss}} := \left(\frac{m-1}{m}\right)^{r \cdot w}.$$

Hence,

$$E_r := \left(\frac{m-1}{m}\right)^{r \cdot w} \cdot m \cdot v \tag{6.1}$$

is the expected number of key candidates that can be sorted out after a single measurement. However, the maximal amount of information an arbitrary number of measurements can reveal is limited by the information leakage. Further measurements will not reveal further information. We verified by experiments that the number of measurements needed to achieve the full information leakage only depends on E_r.

In the sequel, we focus on methods to counteract CBAs. In general, there are two approaches to counteract such a side channel. The first approach is to use some kind of randomization to ensure that the leaking information does not reveal information about the secret key. Using randomization is a general strategy that protects against several kinds of side channel attacks, see for example Chapter 4 (page 25). In Section 6.7 we analyze a more efficient method to counteract CBAs based on random permutations. Before that, we consider the second approach that is to reduce the bandwith of the side channel. We present several implementations of AES and examine their information leakage and their resistance.

6.6 Information Leakage and Resistance of Selected Implementations

As Bernstein pointed out in (Bernstein 2005) to thwart cache attacks it is not sufficient to load all sbox entries into the cache before accessing the sbox in order to compute an intermediate result because $\mathcal{A}$ can get cache information at all times. Hence, loading the complete sbox into the cache does not suffice to hide all cache information. Therefore, he advises to avoid the usage of table lookups in cryptographic algorithms. Computing the AES `SubBytes` operation according to its definition

$$\begin{aligned} f : \{0,1\}^8 &\rightarrow \{0,1\}^8 \\ x &\mapsto a \cdot \mathrm{INV}(x) \oplus b \end{aligned}$$

would virtually cause no cache accesses and hence seems to be secure against CBAs. However, implementing `SubBytes` like this would result in a very inefficient implementation on a PC. To achieve a high level of efficiency people prefer to use precomputed tables. In the sequel, we analyze the security of some well known and some novel variations of implementations of AES. For each of these implementations we consider access driven CBAs based on different sboxes and examine the information leakage and the resistance as defined in (6.1). To simplify notation we fix the size of a cache line to 512 bits as we did above. Furthermore, we did timing experiments for each implemention to estimate its efficiency. The testing environment for our timing experiments is shown in Table 6.3. For each implementation we compare its timing with the timing of the fast implementation. Table 6.9 summarizes the information leakage, resistance and efficiency for all considered implementations.

CPU	Intel Pentium M, 1400MHz
OS	Linux, Kernel 2.6.18
Compiler	gcc 4.1.1

Table 6.3: Experimental environment

Standard Implementation

The standard implementation as described in Section 2.3 (page 9) uses only the standard sbox **S**. Hence, an access driven CBA as described above is based on that sbox. The standard sbox consists of 256 entries each of size one byte. Hence, the sbox can be stored in $m = 4$ cache lines each of which can hold $v = 64$ sbox entries. In each round the sbox is applied $w = 16$ times. Next, we analyze the susceptibility to access driven CBAs as described above:

Information leakage To determine the number of leaking bits we performed experiments. Due to the low number m of cache lines and the relative high number of sbox accesses per round the probability that a cache line is not accessed in a part of the encryption becomes very small with an increasing number r of involved rounds. We verified by experiments that measurements taken over ≤ 3 rounds of the standard implementation leak all key bits. Although the small probability p_{miss} prevents performing further experiments we assume that even more rounds will leak all key bits.

Resistance As explained above, the probability that a cache line is not accessed during r rounds of an encryption decreases rapidly with increasing r. Table 6.4 summarizes the resistance of the standard implementation for $1 \leq r \leq 10$.

r	E_r
1	2.57
2	$2.57 \cdot 10^{-2}$
3	$2.58 \cdot 10^{-4}$
4	$2.58 \cdot 10^{-6}$
5	$2.59 \cdot 10^{-8}$
6	$2.59 \cdot 10^{-10}$
7	$2.60 \cdot 10^{-12}$
8	$2.61 \cdot 10^{-14}$
9	$2.61 \cdot 10^{-16}$
10	$2.62 \cdot 10^{-18}$

Table 6.4: The resistance of the standard implementation

E.g., we expect that a single measurement taken over 2 rounds of the encryption allows to sort out approximately 0.0257 key candidates.

Efficiency The standard implementation uses some time consuming operations such as matrix multiplication over the finite field $\mathbb{F}_{256}$. Hence, on a 32 bit processor the efficiency of the standard implementation is obviously lower than the efficiency of the fast implementation that avoids these inefficient operations. Our timing experiments on a 32 bit processor have shown that the standard implementation is about 3 times slower than the fast implementation.

Fast Implementation

The fast implementation as described in Section 2.4 (page 16) is the reference implementation for virtually all AES implementations in software on 32 bit platforms. Its performance is based on the clever merge of the round functions `SubBytes`, `ShiftRows` and `MixColumns` into 5 specially constructed sboxes $\mathbf{T}_0, \dots, \mathbf{T}_4$. Each of these sboxes holds 256 entries of size 4 bytes. Hence, a cache line can store $v = 16$ sbox elements and we need $m = 16$ cache lines to store an sbox $\mathbf{T}_i$ in the cache. As described above, each of the sboxes $\mathbf{T}_0, \dots, \mathbf{T}_3$ is applied 4 times in every round $1, \dots, 9$ of the encryption. In the last round $\mathbf{T}_4$ is applied 16 times. We consider both, a CBA based on table lookups to one of the sboxes $\mathbf{T}_0, \dots, \mathbf{T}_3$ in the first round like the one described in Section 6.3.1 and a CBA based on the sbox $\mathbf{T}_4$ of the last round as described in Section 6.3.2.

Information leakage The access driven CBA of (Osvik et al. 2006) as described in Section 6.3.1 on the first round of AES shows that in this case the fast implementation will reveal half of the key bits, even with an arbitrary number of measurements. As we have seen in Section 6.3.2 (page 87) a CBA based on the table lookups to $\mathbf{T}_4$ in the last round lets $\mathcal{A}$ determine the secret key completely.

Resistance Due to the bigger size of the sboxes and the lower number of sbox lookups per round the resistance of the fast implementation is significantly lower than that of the standard implementation. If the attack is based on sbox $\mathbf{T}_4$ than every measurement is implicitly restricted to the last round because $\mathbf{T}_4$ is only used in that round. Hence, the resistance does not change for measurements restricted to a different number of rounds. We expect that $\mathcal{A}$ can rule out approximately

$$E_r = \left(\frac{15}{16}\right)^{16} \cdot 16 \cdot 16 \approx 91$$

wrong key candidates of a key byte of the last round key after a single measurement.

If the access driven CBA is based on sbox lookups of the first round things are different. Each sbox $\mathbf{T}_0, \dots, \mathbf{T}_3$ is used 4 times in every round $1, \dots, 9$. In this case, the expected numbers of wrong key candidates that can be ruled out after a single measurement taken over r rounds are given in Table 6.5.

Efficiency As the name suggests, the fast implementation is very efficient especially on 32 bit computers. It only consists of sbox lookups, shifts and XOR operations and omits the complex operations such as matrix multiplication and uses precomputed tables to compute operations in finite fields.

r	E_r
1	198.0
2	153.0
3	118.0
4	91.2
5	70.4
6	54.4
7	42.0
8	32.5
9	25.1

Table 6.5: The resistance of the fast implementation against access driven CBAs based on sboxes $\mathbf{T}_0, \ldots, \mathbf{T}_3$.

Fast Implementation Using Standard Sbox in the Last Round (fast-1)

To improve the security, the authors of (Brickell et al. 2006) suggested to exchange the sbox $\mathbf{T}_4$ with the standard sbox in the last round. In the case of a CBA that is based on sbox lookups of the first round this implementation provides the same information leakage and resistance as the fast implementation. Therefore, in the sequel we only consider a CBA that is based on the table lookups of the last round.

Information leakage As for the standard implementation explained above, an access driven CBA based on the standard sbox used in the last round reveals the complete secret key.

Resistance The resistance of this approach against an access driven CBA based on the standard sbox is better than that of the fast implementation against an access driven CBA based on the sbox $\mathbf{T}_4$. A single measurement lets $\mathcal{A}$ rule out approximately

$$E_r = \left(\frac{3}{4}\right)^{16} \cdot 4 \cdot 64 \approx 2.57$$

wrong key candidates. The resistance remains constant because the standard sbox is only used in one round.

Efficiency Timing experiments with our implementation of this approach showed that using the standard sbox in the last round does not slow down the encryption significantly.

Fast Implementation Using only Sbox $\mathbf{T}_0$ (fast-2)

We consider another modification of the fast implementation of AES. The description of AES in Section 2.4 (page 16) shows that the i-th entry of the sboxes $\mathbf{T}_1, \ldots, \mathbf{T}_3$ is equal to the i-th entry of the sbox $\mathbf{T}_0$ cyclically shifted by $1, 2$ and 3 bytes to the right respectively. Hence, we propose to use only sbox $\mathbf{T}_0$ in the encryption and shift the result as needed to compute the correct AES encryption. E.g., to compute the sbox lookup $\mathbf{T}_1[i]$ using the sbox $\mathbf{T}_0$ we simply cyclically shift the value $\mathbf{T}_0[i]$ by 1 byte to the right. In the last round, we recommend to use the standard sbox. Since we already analyzed the information leakage and resistance of the standard sbox we focus on a CBA based on the sbox $\mathbf{T}_0$.

Information leakage Using only the sbox $\mathbf{T}_0$ does not change the amount of information that leaks compared to the fast implementation. Hence, this implementation causes also the leakage of the complete secret key.

Resistance The sbox $\mathbf{T}_0$ needs $m = 16$ cache lines each of which stores $v = 16$ elements. The difference with the fast implementation is that $\mathbf{T}_0$ is applied $w = 16$ times in each round $1, \ldots, 10$. Due to the increased number of sbox lookups per round the resistance against access driven CBAs is better than the resistance of the fast implementation. Table 6.6 (page 96) shows the resistance E_r for all different values r.

r	E_r
1	91.2
2	32.5
3	11.6
4	4.12
5	1.47
6	$5.22 \cdot 10^{-1}$
7	$1.86 \cdot 10^{-1}$
8	$6.62 \cdot 10^{-2}$
9	$2.36 \cdot 10^{-2}$
10	$8.39 \cdot 10^{-3}$

Table 6.6: The resistance of the fast implementation using only $\mathbf{T}_0$

Efficiency We implemented this approach and did timing measurements to estimate the running time. Compared to the fast implementation we could not measure any differences in the running time. Hence, this implementation is as efficient as the fast implementation.

Splitted Sboxes (small-n)

As a simple but effective countermeasure to counteract access driven CBAs we suggest to split the sbox $\mathbf{S}$ into n smaller sboxes $\mathbf{S}_0, \ldots, \mathbf{S}_{n-1}$ such that every small sbox $\mathbf{S}_i$ fits completely into a single cache line[5]. An application $\mathbf{S}_i[x]$ of sbox $\mathbf{S}_i$ yields d_i bits of the desired result $\mathbf{S}[x]$. Hence, the correct result can be calculated by computing all bits separately and shift them into the correct position.

We construct the small sboxes $\mathbf{S}_i$ for $0 \leq i \leq n-1$ as follows:

$$\mathbf{S}_i : \{0,1\}^8 \to \{0,1\}^{d_i}$$

mapping

$$x \mapsto \lfloor\ \mathbf{S}[x]\ \rfloor_{\left(\sum_{j=0}^{i-1} d_j, \left(\sum_{j=0}^{i} d_j\right)-1\right)}$$

where $\lfloor\ y\ \rfloor_{(b,e)}$ are the bits $y_b \dots y_e$ of the binary representation of $y = (y_0, \dots, y_7)$. The small sboxes are shown in Appendix B (page 115). Instead of applying the sbox $\mathbf{S}$ to x directly each $\mathbf{S}_i$ is applied.

The result is computed as

$$\mathbf{S}[x] = \sum_{i=0}^{n-1} \mathbf{S}_i[x] \cdot 2^{\sum_{j=0}^{i-1} d_j}.$$

In the sequel, we assume that the size of the sbox is a multiple of the size of a cache line and that all d_j are equal. Depending on the number of small sboxes we call this implementation *small-n*. E.g., let the size of a cache line be $\lambda = 512$ bits and for $0 \leq i \leq 3$ let each $\mathbf{S}_i$ store the bits $\lfloor\ \mathbf{S}[x]\ \rfloor_{(2i,2i+1)}$. The result $\mathbf{S}[x]$ is then computed as

$$\mathbf{S}[x] = \mathbf{S}_0[x] \oplus \mathbf{S}_1[x] \cdot 4 \oplus \mathbf{S}_2[x] \cdot 16 \oplus \mathbf{S}_3[x] \cdot 64.$$

We call this implementation *small*-4.

Information leakage The amount of information that leaks depends on the number n of small sboxes. Let us consider the variants small-2, small-4 and small-8. Computing $\mathbf{S}[x]$ using variant small-4 or small-8 leaks 0 bits of information having cache lines of size 512 bits because of two reasons:

1. Every $\mathbf{S}_i$ fits completely into a single cache line.
2. For every x each $\mathbf{S}_i$ is used exactly once to compute $\mathbf{S}[x]$.

Hence, the cache information remains constant for all inputs. An attacker will always get the information that every cache line has been accessed even if he could restrict measurements to single sbox lookups. The only assumption that is involved is that $\mathcal{A}$ cannot distinguish between the accesses on different elements within the same cache line (Section 6.2.4). The variant small-2 presumably leaks all key bits in our setting.

Resistance As we have shown above, the variants small-4 and small-8 leak no key bit. Hence, even an arbitrary number of measurements does not provide any information that lets $\mathcal{A}$ restrict the number of possible keys. This implies that small-4 and small-8 have resistance 0. The resistance of small-2 is listed in Table 6.7.

[5] Each sbox should fit into a single cache line at every cache level.

r	E_r
1	$3.91 \cdot 10^{-3}$
2	$5.96 \cdot 10^{-8}$
3	$9.09 \cdot 10^{-13}$
4	$1.39 \cdot 10^{-17}$
5	$2.12 \cdot 10^{-22}$
6	$3.23 \cdot 10^{-27}$
7	$4.93 \cdot 10^{-32}$
8	$7.52 \cdot 10^{-37}$
9	$1.15 \cdot 10^{-41}$
10	$1.75 \cdot 10^{-46}$

Table 6.7: The resistance of small-2

Efficiency Obviously, the performance depends on the number of involved sboxes and shifts to move bits into the right position. To estimate the efficiency we used the small-n variants in the last round of the fast implementation. Due to the inefficient bit manipulations on 32 bit processors our ad hoc implementation of using small-4 only in the last round shows that the penalty is about 60%. We expect that a more sophisticated implementation reduces this penalty significantly. However, we stress that access driven CBAs are very powerful attacks. Hence, it is not astonishing that secure implementations are not that efficient.

Table 6.8 shows the result of our timing measurements for the variants small-2, small-4 and small-8 applied only on the last round of the fast implementation of AES. Applying the small variants to more rounds will decrease the efficiency further.

Comparison of Implementations

To compare the implementations considered above with respect to information leakage (IL), resistance (E_r) and efficiency (Eff.) we summarize the important information in Table 6.9. The explanations of the detailed informations were given above.

# sboxes	fast	small-2	small-4	small-8
time factor	1	1.32	1.6	1.95

Table 6.8: Timings for small-2, small-4 and small-8 applied on the last round of AES

	1	2	3	4	5	6	7	8
	standard	fast		fast-1	fast-2	small-2	small-4	small-8
	$\mathbf{S}$	$\mathbf{T}_0, \ldots, \mathbf{T}_3$	$\mathbf{T}_4$	$\mathbf{S}$	$\mathbf{T}_0$	$\mathbf{S}_0, \mathbf{S}_1$	$\mathbf{S}_0, \ldots, \mathbf{S}_3$	$\mathbf{S}_0, \ldots, \mathbf{S}_7$
IL	8	4/8	8	8	8	8	0	0
E_1	2.57	198.0	91.2	2.57	91.2	$3.91 \cdot 10^{-3}$	0	0
E_2	$2.57 \cdot 10^{-2}$	153.0	91.2	2.57	32.5	$5.96 \cdot 10^{-8}$	0	0
E_3	$2.58 \cdot 10^{-4}$	118.0	91.2	2.57	11.6	$9.09 \cdot 10^{-13}$	0	0
E_4	$2.58 \cdot 10^{-6}$	91.2	91.2	2.57	4.12	$1.39 \cdot 10^{-17}$	0	0
E_5	$2.59 \cdot 10^{-8}$	70.4	91.2	2.57	1.47	$2.12 \cdot 10^{-22}$	0	0
E_6	$2.59 \cdot 10^{-10}$	54.4	91.2	2.57	$5.22 \cdot 10^{-1}$	$3.23 \cdot 10^{-27}$	0	0
E_7	$2.60 \cdot 10^{-12}$	42.0	91.2	2.57	$1.86 \cdot 10^{-1}$	$4.93 \cdot 10^{-32}$	0	0
E_8	$2.61 \cdot 10^{-14}$	32.5	91.2	2.57	$6.62 \cdot 10^{-2}$	$7.52 \cdot 10^{-37}$	0	0
E_9	$2.61 \cdot 10^{-16}$	25.1	91.2	2.57	$2.36 \cdot 10^{-2}$	$1.15 \cdot 10^{-41}$	0	0
E_{10}	$2.62 \cdot 10^{-18}$	25.1	91.2	2.57	$8.39 \cdot 10^{-3}$	$1.75 \cdot 10^{-46}$	0	0
Eff.	~ 3	1		~ 1	~ 1	1.32	1.6	1.95

Table 6.9: Comparison of selected AES implementations with respect to information leakage (IL), resistance (E_r) and efficiency (Eff.)

The standard implementation leaks all key bits and provides good resistance but low efficiency. The fast implementation also leaks all key bits and provides low resistance but good efficiency. The modifications fast-1 and fast-2 inherit the information leakage and the good efficiency but improve the resistance. Fast-1 improves the resistance against CBAs that are based on the last round from 91.2 to 2.57. Using the fast-1 implementation a CBA based on the sboxes $\mathbf{T}_0, \ldots, \mathbf{T}_3$ is much more efficient than a CBA based on the last round. Fast-2 uses only one large sbox and hence improves the resistance against all CBAs that comply with our basic structure of access driven CBAs. As the implementations mentioned above, the implementation small-2 leaks all key bits. Its resistance is much better than the resistance of the implementations mentioned above but its efficiency is rather low. The implementations small-4 and small-8 do not leak a single key bit and hence provide the best possible resistance. As the implementation small-2, the implementations small-4 and small-8 suffer from low efficiency. See Table 6.10 for a simplified comparison of the implementations considered above.

For applications that require high speed we propose to use the implementation fast-2 because its efficiency is comparable to the efficiency of the fast implementation. However, one should keep in mind that fast-2 does not thwart access driven CBAs completely but only increase the complexity of a CBA. In high security applications where it is inevitable to thwart CBAs we propose to use the small-4 implementation. It suffers from rather low efficiency but prevents the leakage of key bits.

implementation	info leakage	resistance	efficiency
standard	8 bit / Byte	+	−
fast	8 bit / Byte	−	+
fast-1	8 bit / Byte	0	+
fast-2	8 bit / Byte	+	+
small-2	8 bit / Byte	++	−−
small-4	0 bit / Byte	++	−−
small-8	0 bit / Byte	++	−−

Table 6.10: Simplified Comparison of Implementations

6.7 Countermeasures Based on Permutations

Another class of countermeasure that was already proposed but not analyzed in (Brickell et al. 2006) is to use secret random permutations to randomize the accesses to the sbox. In this section we present a CBA against an implementation of AES secured by a random permutation that needs roughly 2300 measurements to reveal the complete key (Blömer and Krummel 2007). This shows that the increase of the complexity of CBAs induced by random permutations is not as high as one would expect. In particular, the uncertainty of the permutation is not a good measure to estimate the gain of security. A random permutation has uncertainty of $\log_2(256!) \approx 1684$ bits and the uncertainty of the induced partition on the cache lines is $\log_2(256!/(16!)^{16}) \approx 976$ bits.

On the other hand, we present a subset of permutations, so called distinguished permutations, that reduce the information leakage from 8 bits to 4 bits per key byte. Hence, the remaining bits must be determined by an additional attack thereby increasing the complexity. In our standard scenario this is the best one can achieve.

We focus only on the protection of the last round of AES and we assume that the output x of the 9th round is randomized using some secret random permutation π. To be more precise, each byte x_i of the state $x = x_0, \ldots, x_{15}$ is substituted by $\pi(x_i)$. To execute the last round of AES a modified sbox $\mathbf{T}'_4$ that depends on π fulfilling

$$\mathbf{T}'_4[\pi(x_i)] = \mathbf{T}_4[x_i]$$

is applied to every byte x_i. This ensures that the resulting ciphertext $c = c_0, \ldots, c_{15}$ is correct. We denote the ℓ-th cache line used for the table lookups for $\mathbf{T}'_4$ by $CL_\ell, \ell = 0, \ldots, 15$. Hence, CL_ℓ contains the 4-tuples

$$\{(\mathbf{S}[\pi^{-1}(x)], \mathbf{S}[\pi^{-1}(x)], \mathbf{S}[\pi^{-1}(x)], \mathbf{S}[\pi^{-1}(x)]) \mid x = 16 \cdot \ell, \ldots, 16 \cdot \ell + 15\}.$$

Using a permutation π, information leaking through accessed cache lines does not depend directly on x_i but only on the permuted value $\pi(x_i)$. Since π is unknown to $\mathcal{A}$ the application of π prevents him to deduce information about the last round key $k^{10} = k_0^{10}, \ldots, k_{15}^{10}$ directly. However, in the sequel we will show how to bypass random permutations by using CBAs.

6.7.1 An Access Driven CBA on a Permuted Sbox

We assume that we have a fast implementation of AES that is protected by a random permutation π as described above. We also assume that the adversary $\mathcal{A}$ has access to the AES decryption algorithm. This assumption can be avoided. However, the exposition becomes easier if we allow $\mathcal{A}$ access to the decryption. We show how an adversary $\mathcal{A}$ can compute the bytes $k_0^{10}, \ldots, k_{15}^{10}$ of the last round key.

Let $\widehat{k}_0$ denote a candidate for byte k_0^{10} of the last round key. In a first step for each possible value $\widehat{k}_0$ the adversary $\mathcal{A}$ determines the assignment $P_{\widehat{k}_0}$ of bytes to cache lines induced by π under the assumption that $\widehat{k}_0 = k_0^{10}$. To be more precise $\mathcal{A}$ computes a function

$$P_{\widehat{k}_0} : \{0,1\}^8 \to \{0, \ldots, 15\}$$

such that if $\widehat{k}_0$ is correct then for all x:

$$\pi(x) \in \{16 \cdot P_{\widehat{k}_0}(x), \ldots, 16 \cdot P_{\widehat{k}_0}(x) + 15\}.$$

I.e., if $\widehat{k}_0$ is correct then $P_{\widehat{k}_0}$ is the correct assignment of values $\pi(x)$ to cache lines.

Let us fix some x and a candidate $\widehat{k}_0$ for k_0^{10}. We set $c_0 = \mathbf{S}[x] \oplus \widehat{k}_0$ and let $\widehat{M}_0 = \{0, \ldots, 15\}$ denote the set of indices of possible cache lines. The adversary $\mathcal{A}$ repeats the following steps for $j = 1, 2, \ldots, n$ until $\widehat{M}_0$ contains a single element.

1. $\mathcal{A}$ chooses a ciphertext c^j, whose first byte is c_0, while the remaining bytes of c^j are chosen independently and uniformly at random.

2. Using his access to the decryption algorithm, $\mathcal{A}$ computes the plaintext p^j corresponding to the c^j.

3. $\mathcal{A}$ triggers an encryption of p^j by the crypto process and obtains cache information. I.e., $\mathcal{A}$ obtains the set D_0^j of cache lines that were accessed when applying sbox T_4' during the encryption of p^j.

4. $\mathcal{A}$ sets $\widehat{M}_0 := \widehat{M}_0 \cap D_0^j$.

If $\widehat{M}_0 = \{y\}$, then $\mathcal{A}$ sets $P_{\widehat{k}_0}(x) = y$. Repeating this process for all x yields the function $P_{\widehat{k}_0}$ which has the desired property.

Under the assumption that the guess $\widehat{k}_0$ was correct, the function $P_{\widehat{k}_0}$ is the correct partition of values $\pi(x)$ into cache lines. Remember that the permutation π is also used to scramble the bytes on the other positions. In particular, the mapping of bytes to cache lines is the same for all positions of the state. Hence, it is not difficult to see that the information provided by $P_{\widehat{k}_0}$ enables the adversary to mount a CBA on the last round similar to the one described in Section 6.3.2 (page 87). This attack can be used to determine for each possible candidate $\widehat{k}_0$ a set of vectors $\widehat{k}_1, \ldots, \widehat{k}_{15}$ of hypotheses for the other key bytes. To determine a candidate $\widehat{k}_i$ that arises from the value of $\widehat{k}_0$ the attacker $\mathcal{A}$ performs the following steps:

1. $\mathcal{A}$ chooses $n \in \mathbb{N}$ plaintexts $p^{(1)}, \ldots, p^{(n)}$
2. $\mathcal{A}$ obtains the ciphertexts and the measurements $m^{(j)} = (D_0^{(j)}, D_1^{(j)}, c^{(j)})$ for $1 \leq j \leq n$.
3. Let x_i denote the i-th byte of the intermediate state after the 9-th round. $\mathcal{A}$ concludes that
$$x_i \in \widehat{X}_i^{(j)} = \bigcup_{\ell \in D_0^{(j)}} \{\widehat{x}_i \mid P_{\widehat{k}_0}(\widehat{x}_i) = \ell\}$$
4. $\mathcal{A}$ computes the sets.
$$\widehat{K}_i^{(j)} = \left\{ c_i^{(j)} \oplus \mathbf{S}\left[\widehat{x}_i^{(j)}\right] \mid \widehat{x}_i^{(j)} \in \widehat{X}_i^{(j)} \right\}$$
for all $1 \leq j \leq n$.
5. $\mathcal{A}$ computes the set
$$\widehat{K}_i = \bigcap_{j=1}^{n} \widehat{K}_i^{(j)}$$
of candidates for k_i.

For the time being, we assume that π has the property that for each $\widehat{k}_0$ there remains only a single vector of hypotheses for the other key bytes. Hence, in the end there are only 256 AES keys left and a simple brute force attack reveals the correct one. In general, a random permutation has this property. For a mathematical precise definition and analysis of that property see Section 6.7.2.

Cost Analysis Experiments show that in the first step of the attack $\mathcal{A}$ needs on average 9 measurements consisting of a pair (p^i, c^i) and the corresponding cache information D_0^i such that the intersection $\widehat{M_0} := \bigcap D_0^i$ contains only a single element $y = P_{\widehat{k}_0}(x)$. We need to determine the mapping $P_{\widehat{k}_0}(x)$ for every key candidate $\widehat{k}_0$ and every argument $x \in \{0,1\}^8$. Hence, a straightforward implementation of the attack needs roughly $256 \cdot 256 \cdot 9$ measurements to determine the function $P_{\widehat{k}_0}(x)$ for all arguments $x \in \{0,1\}^8$ and all key candidates $\widehat{k}_0 \in \{0,1\}^8$. However, one can reuse measurements for different key candidates $\widehat{k}_0, \widehat{k}_0'$ to reduce the number of measurements to roughly $256 \cdot 9 = 2304$. To determine the vector of hypothesis based on the candidate $\widehat{k}_0$ we can reuse the measurements obtained by determining the function $P_{\widehat{k}_0}$. Hence, the expected number of measurements of this attack is 2304.

6.7.2 Separability and Distinguished Permutations

From a security point of view, it is desirable to reduce the information leakage. E.g., a cache attack alone should reveal as few information as possible, in particular it should not reveal the complete key. Then the adversary is forced to either mount a refined and more complex CBA based on other intermediate results or combine the cache attack with some other method to determine the key bytes uniquely. In this case, the situation is similar to the attack of (Osvik et al. 2006), where a cache attack on the first round only reveals 4 bits of each key byte. Hence Osvik et al. combine cache attacks on the first and second round of AES.

First, we present the property a permutation applied to the result of the 9-th round should have such that $\mathcal{A}$ cannot determine the key bytes uniquely using only a cache attack on the last round. We denote the ℓ-th cache line by CL_ℓ and the elements of CL_ℓ by $a_0^{(\ell)}, \ldots, a_{15}^{(\ell)}$. Hence, the underlying permutation used to define this cache line is given by

$$\pi^{-1}(16\ell + j) = \mathbf{S}^{-1}[a_j^{(\ell)}] \tag{6.2}$$

for $j = 0, \ldots, 15$.

We say that a key candidate $\widehat{k}_0$ is *separable* from the first key byte k_0^{10} of the last round if there exists a measurement that proves $\widehat{k}_0$ to be wrong. Conversely, a key candidate $\widehat{k}_0$ is *inseparable* from the key k_0^{10} if there does not exist a measurement that proves $\widehat{k}_0$ to be wrong. More precisely, writing $\widehat{k}_0 = k_0^{10} \oplus \delta$ the bytes $\widehat{k}_0$ and $k_0^{10} \oplus \delta$ are inseparable if and only if

$$\forall \ell \in \{0, \ldots, 15\} \forall a \in CL_\ell : a \oplus \delta \in CL_\ell. \tag{6.3}$$

Notice that this property only depends on the difference δ and not on the value of k_0. In our setting there are 16 elements of the sbox in every cache line and therefore property (6.3) can only be satisfied by at most 16 differences.

It turns out that for $|\Delta| = 16$ the set

$$\Delta := \{\delta \mid \text{for all } k_0 \in \{0,1\}^8 \text{ the bytes } k_0 \text{ and } k_0 \oplus \delta \text{ are inseparable}\}$$

forms a 4 dimensional subspace of $\mathbb{F}_{2^8}$ viewed as a 8 dimensional vector space over $\mathbb{F}_2$. It is obvious that the neutral element 0 is an element of Δ and that every $\delta \in \Delta$ is its own inverse. It remains to show that Δ is closed with respect to addition. Consider $\delta, \delta' \in \Delta$ and an arbitrary $a \in CL_\ell$. Then $a' = a \oplus \delta \in CL_\ell$ implies that $a' \oplus \delta' = a \oplus \delta \oplus \delta' \in CL_\ell$ because of (6.3) and $\delta \oplus \delta' \in \Delta$ holds.

Hence, any partition that has the maximal number of inseparable key candidates must generate a subspace of dimension 4.

Using this observation we describe how to efficiently construct permutations such that the set Δ of inseparable differences has size 16. In the sequel, we will call any such permutation a *distinguished permutation*.

Construction of the Subspace We first construct a set Δ of 16 differences that is closed with respect to addition over $\mathbb{F}_{256}$. We can do this in the following way

1. set $\Delta := \{\delta_0 := 0\}$, choose δ_1 uniformly at random from the set $\{1, \dots, 255\}$, set $\Delta := \Delta \cup \{\delta_1\}$
2. choose δ_2 uniformly at random from $\{1, \dots, 255\} \setminus \Delta$, set $\Delta := \Delta \cup \{\delta_2, \delta_3 := \delta_1 \oplus \delta_2\}$
3. choose δ_4 uniformly at random from $\{1, \dots, 255\} \setminus \Delta$, set $\Delta := \Delta \cup \{\delta_4, \delta_5 := \delta_4 \oplus \delta_1, \delta_6 := \delta_4 \oplus \delta_2, \delta_7 := \delta_4 \oplus \delta_3\}$
4. choose δ_8 uniformly at random from $\{1, \dots, 255\} \setminus \Delta$, set $\Delta := \Delta \cup \{\delta_8, \delta_9 := \delta_8 \oplus \delta_1, \delta_{10} := \delta_8 \oplus \delta_2, \delta_{11} := \delta_8 \oplus \delta_3, \delta_{12} := \delta_8 \oplus \delta_4, \delta_{13} := \delta_8 \oplus \delta_5, \delta_{14} := \delta_8 \oplus \delta_6, \delta_{15} := \delta_8 \oplus \delta_7\}$

This construction ensures that Δ is closed with respect to addition and hence Δ forms a subspace as desired.

Construction of the Permutation Now we can compute the function P that maps $\mathbf{S}[x] \in \mathbb{F}_2^8$ to a cache line. We use the fact that 16 proper translations of a 4 dimensional subspace form a partition of a 8 dimensional vector space $\mathbb{F}_2^8$. A basis $\{b_0, \dots b_3\}$ of the subspace Δ can be expanded by 4 vectors $b_4, \dots b_7$ to a basis of $\mathbb{F}_2^8$. The 16 translations of Δ generated by linear combinations of $b_4, \dots, b_7$ form the quotient space $\mathbb{F}_2^8/\Delta$ that is a partition of $\mathbb{F}_2^8$. To construct the function P we do the following:

1. for every cache line CL_ℓ do
2. choose $a^{(\ell)}$ uniformly at random from $\mathbb{F}_{256}/\{a^{(j)} \oplus \delta \mid j < \ell, \delta \in \Delta\}$
3. fill CL_ℓ with the values of the set $\{a^{(\ell)} \oplus \delta \mid \delta \in \Delta\}$

Using (6.2) this partition into cache lines defines the corresponding permutation.

Analysis of the Countermeasure The security using a distinguished permutation as defined above rests on two facts.

1. Using a distinguished permutation where the set Δ of inseparable differences has size 16, a cache attack on the last round of AES will reveal only four bits of each key byte k_i^{10}. Overall 64 of the 128 bits of the last round key remain unknown. Therefore, the adversary has to combine his cache attack on the last round with some other method to determine the remaining 64 unknown bits. For example, he could try a modified cache attack on the 9-th round exploiting his partial knowledge of the last round key. Or he could use a brute force search to determine the last round key completely.

2. There are several distinguished permutations and each of these permutations leads to 16! different functions mapping elements to 16 lines. If we choose randomly one of these functions, before an adversary can mount a cache attack on the last round as described in Section 6.3.2, he first has to use some method like the one described in Section 6.7.1 to determine the function P that is actually used.

We stress that we consider the first fact to be the more important security feature. We saw already in Section 6.7.1 that determining a random permutation used for mapping elements to cache lines is not as secure as one might expect. Since we are using permutations of a special form the attack described in Section 6.7.1 can be improved somewhat. In the remainder of this section we briefly describe this improvement. To do so, first we have to determine the number of subspaces leading to distinguished permutations.

As before view $\mathbb{F}_2^n := \{0,1\}^n$ as an n-dimensional $\mathbb{F}_2$ vector space. For $0 \leq k \leq n$ we define $D_{n,k}$ to be the number of k-dimensional subspaces of $\mathbb{F}_2^n$. To determine $D_{n,k}$ for V an arbitrary m-dimensional subspace of $\mathbb{F}_2^n$ we define

$$N_{m,k} := |\{(v_1, \ldots, v_k) | v_i \in V, v_1, \ldots v_k \text{ are linearly independent}\}|.$$

The number $N_{m,k}$ is independent of the particular m-dimensional subspace V, it only depends on the two parameters m and k. Then

$$D_{n,k} = \frac{N_{n,k}}{N_{k,k}}.$$

Next we observe that

$$N_{m,k} = \prod_{j=0}^{k-1} (2^m - 2^j) = 2^{k(k-1)/2} \prod_{j=0}^{k-1} (2^{m-j} - 1).$$

Hence, we obtain that

$$D_{n,k} = \frac{\prod_{j=0}^{k-1} (2^{n-j} - 1)}{\prod_{j=0}^{k-1} (2^{k-j} - 1)}.$$

In our special case we have $n = 8$ and $k = 4$ and hence the number of 4 dimensional subspaces is

$$D_{8,4} = \frac{255 \cdot 127 \cdot 63 \cdot 31}{15 \cdot 7 \cdot 3 \cdot 1} = 200787.$$

As mentioned above, each subspace leads to 16! different distinguished permutations. Hence, overall we have $200787 \cdot 16! \approx 2^{60}$ distinguished permutations. On the other hand, because of the special structure of our permutations, to determine the function P by cache attacks can be done more efficiently than determining an arbitrary function mapping elements to cache lines (see Section 6.7.1). In particular, $\mathcal{A}$ only needs to observe about 7 accesses of a single but arbitrary cache line. With high probability this will be enough to determine a basis of the subspace being used. In addition, $\mathcal{A}$ needs at least one access for every other cache line in order to determine the function P. The corresponding probability experiment follows the multinomial distribution. We did not calculate the expected number of tries exactly. Experiments show that if we can determine the accessed cache line exactly, on average 62 measurements suffice to compute the function P exactly. However, a single measurement only yields a set of accessed cache lines. But arguments similar to the ones used for the first part of the attack in Section 6.7.1 show that we need on average 9 measurements to uniquely determine an accessed cache line. Therefore, on average we need $9 \cdot 62 = 558$ experiments to determine the function P.

Hence, compared to the results of Section 6.7.1 we have reduced the number of measurements used to determine the function P by a factor of 3. However, we want to stress again, that the main security enhancement of using distinguished permutations instead of arbitrary permutations is the fact, that with distinguished permutations the last round key cannot be determined by a cache attack on the last round alone. To improve the security, one can choose larger key sizes such as 192 bits or 256 bits. Since distinguished permutations protect half of the key bits, the remaining uncertainty about the secret key after cache attacks can be increased from 64 bits to 96 bits or 128 bits, respectively.

Separability and Random Permutations In our CBA on an implementation protected by a random permutation (Section 6.7.1) we assumed that fixing a candidate $\widehat{k}_0$ determines the candidates for all other key bytes. With sufficiently many measurements for a fixed $\widehat{k}_0$ we can determine the function $P_{\widehat{k}_0}$ as defined in Section 6.7.1. Furthermore, we saw that the separability of candidates $\widehat{k}, \widehat{k}'$ depends only on their difference $\delta = \widehat{k} \oplus \widehat{k}'$. Hence, to be able to rule out all but one candidate $\widehat{k}_i$ at position i for a fixed $\widehat{k}_0$ the permutation π must have the following property:

$$\forall \delta \neq 0 \exists j \in \{0, \dots, 15\} \exists a \in CL_j : a \oplus \delta \notin CL_j.$$

There are approximately 2^{844} of the $256! \approx 2^{1684}$ permutations that do not have this property. Hence, a random permutation satisfies this condition with probability $1 - \frac{2^{884}}{2^{1684}} = 1 - 2^{-840}$.

6.8 Summary of Countermeasures and Open Problems

In this chapter we presented and analyzed the security of several different implementations of AES. Moreover, we analyzed countermeasures based on permutations: random permutations and distinguished permutations. We give a short overview over the advantages and disadvantages of the countermeasures:

countermeasure	# measurements	information leakage	security	efficiency
small-4	∞	0 bits	high	slow
random permutation	2300	128 bits	low	fast
distinguished permutations	560	64 bits	medium	fast

The second column shows the expected number of measurements an attacker has to perform in order to get the amount of information shown in the third column.

Small-4 (see Section 6.6) prevents information leakage in a cache attack. However, the efficiency depends on the size of a cache line and is rather low. In contrast, random permutations (see Section 6.7) provide only low security. About 2300 measurements are sufficient to reveal the complete 128 bit AES key. If realized via table lookups, random permutations are fast. But to increase the security offered by random permutations they have to be changed frequently. Changing a permutation may cause problems with respect to efficiency and security. So far, we have no precise analysis of these issues.

Distinguished permutations (see Section 6.7.2) protect half of the key bits and hence provide a medium level of security. Using distinguished permutations, no frequent changes of permutations are required to achieve a medium level of security. Hence, they do not suffer from the above mentioned problems of random permutations. Therefore, distinguished permutations provide a better ratio of efficiency and security as random permutations but still leak half of the key bits.

Random permutations and distinguished permutations have to be realized as tables for efficiency reasons. Hence, a straightforward implementation of the applications of a permutation would render the whole implementation susceptible to cache attacks. A possible solution to this problem is to realize permutations via small sboxes that completely fit into a cache line. Following the description of the small-4 variant of Section 6.6, π is split into smaller tables $\pi_0, \ldots, \pi_3$ each of which is applied to the input x. Obviously, this does not make sense if the standard sbox $\mathbf{S}$ is used because both π and $\mathbf{S}$ map from $\{0,1\}^8$ to $\{0,1\}^8$. Hence, it takes as many table lookups to apply π realized with small sboxes as it takes to apply $\mathbf{S}$ realized with small sbox directly. Moreover, realizing $\mathbf{S}$ via small tables has the advantage of not leaking information via the cache behavior.

The situation is different if the large sboxes of the fast implementation are used. Again π maps from $\{0,1\}^8$ to $\{0,1\}^8$ but a large sbox maps from $\{0,1\}^8$ to $\{0,1\}^{32}$. Therefore, it takes 4 times as many table lookups to realize the large sbox via small sboxes than to realize π via small tables.

Hence, first applying π to an input via small tables and then applying a large permuted sbox, as shown in Figure 6.7, makes sense if this technique is faster than realizing the standard sbox $\mathbf{S}$ via small sboxes. Here, one has to take into account the technical problem that on

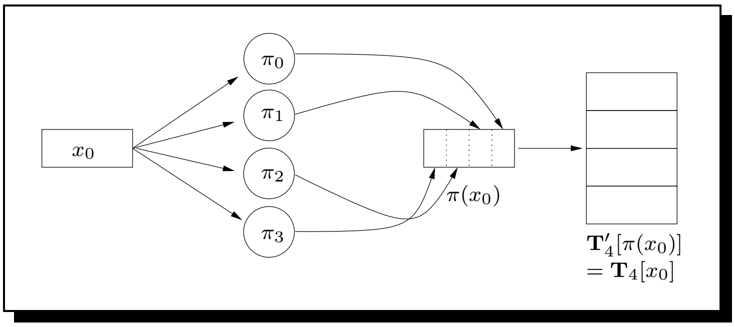

Figure 6.7: Combining small tables with permutation π

32-bit platforms the byte oriented structure of the standard sbox **S** leads to a time consuming post processing to incorporate the output of the sbox into the encryption state.

Note that realizing π via small tables does not leak any information in cache attacks. Only the application of the permuted sbox leaks information about intermediate states. Hence, this scenario is exactly the scenario of our attack in Section 6.7.1 where we assumed that only the application of the sbox leaks information.

As mentioned in Section 6.6 one can scale the sizes of the smaller tables to improve efficiency. But it is essential to determine whether the amount of information that leaks with this method is acceptable or not. Summing up, the analysis given above shows that permutations as a countermeasure to thwart cache based attacks do not provide as much security as one would expect. However, we have shown that using distinguished permutations one can reduce the information leakage via CBAs. That means that even with an arbitrary number of measurements a CBA based on the last round cannot determine certain bits of the secret key. Since we consider the reduction of information leakage as a preferred goal distinguished permutations constitute an interesting way to improve the security gain of permutations.

Appendix A

Sbox Tables $\mathbf{T}_0, \ldots, \mathbf{T}_4$ of AES

	0	1	2	3	4	5	6	7
00	C6 63 63 A5	F8 7C 7C 84	EE 77 77 99	F6 7B 7B 8D	FF F2 F2 0D	D6 6B 6B BD	DE 6F 6F B1	91 C5 C5 54
01	60 30 30 50	02 01 01 03	CE 67 67 A9	56 2B 2B 7D	E7 FE FE 19	B5 D7 D7 62	4D AB AB E6	EC 76 76 9A
02	8F CA CA 45	1F 82 82 9D	89 C9 C9 40	FA 7D 7D 87	EF FA FA 15	B2 59 59 EB	8E 47 47 C9	FB F0 F0 0B
03	41 AD AD EC	B3 D4 D4 67	5F A2 A2 FD	45 AF AF EA	23 9C 9C BF	53 A4 A4 F7	E4 72 72 96	9B C0 C0 5B
04	75 B7 B7 C2	E1 FD FD 1C	3D 93 93 AE	4C 26 26 6A	6C 36 36 5A	7E 3F 3F 41	F5 F7 F7 02	83 CC CC 4F
05	68 34 34 5C	51 A5 A5 F4	D1 E5 E5 34	F9 F1 F1 08	E2 71 71 93	AB D8 D8 73	62 31 31 53	2A 15 15 3F
06	08 04 04 0C	95 C7 C7 52	46 23 23 65	9D C3 C3 5E	30 18 18 28	37 96 96 A1	0A 05 05 0F	2F 9A 9A B5
07	0E 07 07 09	24 12 12 36	1B 80 80 9B	DF E2 E2 3D	CD EB EB 26	4E 27 27 69	7F B2 B2 CD	EA 75 75 9F
08	12 09 09 1B	1D 83 83 9E	58 2C 2C 74	34 1A 1A 2E	36 1B 1B 2D	DC 6E 6E B2	B4 5A 5A EE	5B A0 A0 FB
09	A4 52 52 F6	76 3B 3B 4D	B7 D6 D6 61	7D B3 B3 CE	52 29 29 7B	DD E3 E3 3E	5E 2F 2F 71	13 84 84 97
0A	A6 53 53 F5	B9 D1 D1 68	00 00 00 00	C1 ED ED 2C	40 20 20 60	E3 FC FC 1F	79 B1 B1 C8	B6 5B 5B ED
0B	D4 6A 6A BE	8D CB CB 46	67 BE BE D9	72 39 39 4B	94 4A 4A DE	98 4C 4C D4	B0 58 58 E8	85 CF CF 4A
0C	BB D0 D0 6B	C5 EF EF 2A	4F AA AA E5	ED FB FB 16	86 43 43 C5	9A 4D 4D D7	66 33 33 55	11 85 85 94
0D	8A 45 45 CF	E9 F9 F9 10	04 02 02 06	FE 7F 7F 81	A0 50 50 F0	78 3C 3C 44	25 9F 9F BA	4B A8 A8 E3
0E	A2 51 51 F3	5D A3 A3 FE	80 40 40 C0	05 8F 8F 8A	3F 92 92 AD	21 9D 9D BC	70 38 38 48	F1 F5 F5 04
0F	63 BC BC DF	77 B6 B6 C1	AF DA DA 75	42 21 21 63	20 10 10 30	E5 FF FF 1A	FD F3 F3 0E	BF D2 D2 6D
10	81 CD CD 4C	18 0C 0C 14	26 13 13 35	C3 EC EC 2F	BE 5F 5F E1	35 97 97 A2	88 44 44 CC	2E 17 17 39
11	93 C4 C4 57	55 A7 A7 F2	FC 7E 7E 82	7A 3D 3D 47	C8 64 64 AC	BA 5D 5D E7	32 19 19 2B	E6 73 73 95
12	C0 60 60 A0	19 81 81 98	9E 4F 4F D1	A3 DC DC 7F	44 22 22 66	54 2A 2A 7E	3B 90 90 AB	0B 88 88 83
13	8C 46 46 CA	C7 EE EE 29	6B B8 B8 D3	28 14 14 3C	A7 DE DE 79	BC 5E 5E E2	16 0B 0B 1D	AD DB DB 76
14	DB E0 E0 3B	64 32 32 56	74 3A 3A 4E	14 0A 0A 1E	92 49 49 DB	0C 06 06 0A	48 24 24 6C	B8 5C 5C E4
15	9F C2 C2 5D	BD D3 D3 6E	43 AC AC EF	C4 62 62 A6	39 91 91 A8	31 95 95 A4	D3 E4 E4 37	F2 79 79 8B
16	D5 E7 E7 32	8B C8 C8 43	6E 37 37 59	DA 6D 6D B7	01 8D 8D 8C	B1 D5 D5 64	9C 4E 4E D2	49 A9 A9 E0
17	D8 6C 6C B4	AC 56 56 FA	F3 F4 F4 07	CF EA EA 25	CA 65 65 AF	F4 7A 7A 8E	47 AE AE E9	10 08 08 18
18	6F BA BA D5	F0 78 78 88	4A 25 25 6F	5C 2E 2E 72	38 1C 1C 24	57 A6 A6 F1	73 B4 B4 C7	97 C6 C6 51
19	CB E8 E8 23	A1 DD DD 7C	E8 74 74 9C	3E 1F 1F 21	96 4B 4B DD	61 BD BD DC	0D 8B 8B 86	0F 8A 8A 85
1A	E0 70 70 90	7C 3E 3E 42	71 B5 B5 C4	CC 66 66 AA	90 48 48 D8	06 03 03 05	F7 F6 F6 01	1C 0E 0E 12
1B	C2 61 61 A3	6A 35 35 5F	AE 57 57 F9	69 B9 B9 D0	17 86 86 91	99 C1 C1 58	3A 1D 1D 27	27 9E 9E B9
1C	D9 E1 E1 38	EB F8 F8 13	2B 98 98 B3	22 11 11 33	D2 69 69 BB	A9 D9 D9 70	07 8E 8E 89	33 94 94 A7
1D	2D 9B 9B B6	3C 1E 1E 22	15 87 87 92	C9 E9 E9 20	87 CE CE 49	AA 55 55 FF	50 28 28 78	A5 DF DF 7A
1E	03 8C 8C 8F	59 A1 A1 F8	09 89 89 80	1A 0D 0D 17	65 BF BF DA	D7 E6 E6 31	84 42 42 C6	D0 68 68 B8
1F	82 41 41 C3	29 99 99 B0	5A 2D 2D 77	1E 0F 0F 11	7B B0 B0 CB	A8 54 54 FC	6D BB BB D6	2C 16 16 3A

Table A.1: Sbox $\mathbf{T}_0$

	0	1	2	3	4	5	6	7
00	A5 C6 63 63	84 F8 7C 7C	99 EE 77 77	8D F6 7B 7B	0D FF F2 F2	BD D6 6B 6B	B1 DE 6F 6F	54 91 C5 C5
01	50 60 30 30	03 02 01 01	A9 CE 67 67	7D 56 2B 2B	19 E7 FE FE	62 B5 D7 D7	E6 4D AB AB	9A EC 76 76
02	45 8F CA CA	9D 1F 82 82	40 89 C9 C9	87 FA 7D 7D	15 EF FA FA	EB B2 59 59	C9 8E 47 47	0B FB F0 F0
03	EC 41 AD AD	67 B3 D4 D4	FD 5F A2 A2	EA 45 AF AF	BF 23 9C 9C	F7 53 A4 A4	96 E4 72 72	5B 9B C0 C0
04	C2 75 B7 B7	1C E1 FD FD	AE 3D 93 93	6A 4C 26 26	5A 6C 36 36	41 7E 3F 3F	02 F5 F7 F7	4F 83 CC CC
05	5C 68 34 34	F4 51 A5 A5	34 D1 E5 E5	08 F9 F1 F1	93 E2 71 71	73 AB D8 D8	53 62 31 31	3F 2A 15 15
06	0C 08 04 04	52 95 C7 C7	65 46 23 23	5E 9D C3 C3	28 30 18 18	A1 37 96 96	0F 0A 05 05	B5 2F 9A 9A
07	09 0E 07 07	36 24 12 12	9B 1B 80 80	3D DF E2 E2	26 CD EB EB	69 4E 27 27	CD 7F B2 B2	9F EA 75 75
08	1B 12 09 09	9E 1D 83 83	74 58 2C 2C	2E 34 1A 1A	2D 36 1B 1B	B2 DC 6E 6E	EE B4 5A 5A	FB 5B A0 A0
09	F6 A4 52 52	4D 76 3B 3B	61 B7 D6 D6	CE 7D B3 B3	7B 52 29 29	3E DD E3 E3	71 5E 2F 2F	97 13 84 84
0A	F5 A6 53 53	68 B9 D1 D1	00 00 00 00	2C C1 ED ED	60 40 20 20	1F E3 FC FC	C8 79 B1 B1	ED B6 5B 5B
0B	BE D4 6A 6A	46 8D CB CB	D9 67 BE BE	4B 72 39 39	DE 94 4A 4A	D4 98 4C 4C	E8 B0 58 58	4A 85 CF CF
0C	6B BB D0 D0	2A C5 EF EF	E5 4F AA AA	16 ED FB FB	C5 86 43 43	D7 9A 4D 4D	55 66 33 33	94 11 85 85
0D	CF 8A 45 45	10 E9 F9 F9	06 04 02 02	81 FE 7F 7F	F0 A0 50 50	44 78 3C 3C	BA 25 9F 9F	E3 4B A8 A8
0E	F3 A2 51 51	FE 5D A3 A3	C0 80 40 40	8A 05 8F 8F	AD 3F 92 92	BC 21 9D 9D	48 70 38 38	04 F1 F5 F5
0F	DF 63 BC BC	C1 77 B6 B6	75 AF DA DA	63 42 21 21	30 20 10 10	1A E5 FF FF	0E FD F3 F3	6D BF D2 D2
10	4C 81 CD CD	14 18 0C 0C	35 26 13 13	2F C3 EC EC	E1 BE 5F 5F	A2 35 97 97	CC 88 44 44	39 2E 17 17
11	57 93 C4 C4	F2 55 A7 A7	82 FC 7E 7E	47 7A 3D 3D	AC C8 64 64	E7 BA 5D 5D	2B 32 19 19	95 E6 73 73
12	A0 C0 60 60	98 19 81 81	D1 9E 4F 4F	7F A3 DC DC	66 44 22 22	7E 54 2A 2A	AB 3B 90 90	83 0B 88 88
13	CA 8C 46 46	29 C7 EE EE	D3 6B B8 B8	3C 28 14 14	79 A7 DE DE	E2 BC 5E 5E	1D 16 0B 0B	76 AD DB DB
14	3B DB E0 E0	56 64 32 32	4E 74 3A 3A	1E 14 0A 0A	DB 92 49 49	0A 0C 06 06	6C 48 24 24	E4 B8 5C 5C
15	5D 9F C2 C2	6E BD D3 D3	EF 43 AC AC	A6 C4 62 62	A8 39 91 91	A4 31 95 95	37 D3 E4 E4	8B F2 79 79
16	32 D5 E7 E7	43 8B C8 C8	59 6E 37 37	B7 DA 6D 6D	8C 01 8D 8D	64 B1 D5 D5	D2 9C 4E 4E	E0 49 A9 A9
17	B4 D8 6C 6C	FA AC 56 56	07 F3 F4 F4	25 CF EA EA	AF CA 65 65	8E F4 7A 7A	E9 47 AE AE	18 10 08 08
18	D5 6F BA BA	88 F0 78 78	6F 4A 25 25	72 5C 2E 2E	24 38 1C 1C	F1 57 A6 A6	C7 73 B4 B4	51 97 C6 C6
19	23 CB E8 E8	7C A1 DD DD	9C E8 74 74	21 3E 1F 1F	DD 96 4B 4B	DC 61 BD BD	86 0D 8B 8B	85 0F 8A 8A
1A	90 E0 70 70	42 7C 3E 3E	C4 71 B5 B5	AA CC 66 66	D8 90 48 48	05 06 03 03	01 F7 F6 F6	12 1C 0E 0E
1B	A3 C2 61 61	5F 6A 35 35	F9 AE 57 57	D0 69 B9 B9	91 17 86 86	58 99 C1 C1	27 3A 1D 1D	B9 27 9E 9E
1C	38 D9 E1 E1	13 EB F8 F8	B3 2B 98 98	33 22 11 11	BB D2 69 69	70 A9 D9 D9	89 07 8E 8E	A7 33 94 94
1D	B6 2D 9B 9B	22 3C 1E 1E	92 15 87 87	20 C9 E9 E9	49 87 CE CE	FF AA 55 55	78 50 28 28	7A A5 DF DF
1E	8F 03 8C 8C	F8 59 A1 A1	80 09 89 89	17 1A 0D 0D	DA 65 BF BF	31 D7 E6 E6	C6 84 42 42	B8 D0 68 68
1F	C3 82 41 41	B0 29 99 99	77 5A 2D 2D	11 1E 0F 0F	CB 7B B0 B0	FC A8 54 54	D6 6D BB BB	3A 2C 16 16

Table A.2: Sbox $\mathbf{T}_1$

	0	1	2	3	4	5	6	7
00	63 A5 C6 63	7C 84 F8 7C	77 99 EE 77	7B 8D F6 7B	F2 0D FF F2	6B BD D6 6B	6F B1 DE 6F	C5 54 91 C5
01	30 50 60 30	01 03 02 01	67 A9 CE 67	2B 7D 56 2B	FE 19 E7 FE	D7 62 B5 D7	AB E6 4D AB	76 9A EC 76
02	CA 45 8F CA	82 9D 1F 82	C9 40 89 C9	7D 87 FA 7D	FA 15 EF FA	59 EB B2 59	47 C9 8E 47	F0 0B FB F0
03	AD EC 41 AD	D4 67 B3 D4	A2 FD 5F A2	AF EA 45 AF	9C BF 23 9C	A4 F7 53 A4	72 96 E4 72	C0 5B 9B C0
04	B7 C2 75 B7	FD 1C E1 FD	93 AE 3D 93	26 6A 4C 26	36 5A 6C 36	3F 41 7E 3F	F7 02 F5 F7	CC 4F 83 CC
05	34 5C 68 34	A5 F4 51 A5	E5 34 D1 E5	F1 08 F9 F1	71 93 E2 71	D8 73 AB D8	31 53 62 31	15 3F 2A 15
06	04 0C 08 04	C7 52 95 C7	23 65 46 23	C3 5E 9D C3	18 28 30 18	96 A1 37 96	05 0F 0A 05	9A B5 2F 9A
07	07 09 0E 07	12 36 24 12	80 9B 1B 80	E2 3D DF E2	EB 26 CD EB	27 69 4E 27	B2 CD 7F B2	75 9F EA 75
08	09 1B 12 09	83 9E 1D 83	2C 74 58 2C	1A 2E 34 1A	1B 2D 36 1B	6E B2 DC 6E	5A EE B4 5A	A0 FB 5B A0
09	52 F6 A4 52	3B 4D 76 3B	D6 61 B7 D6	B3 CE 7D B3	29 7B 52 29	E3 3E DD E3	2F 71 5E 2F	84 97 13 84
0A	53 F5 A6 53	D1 68 B9 D1	00 00 00 00	ED 2C C1 ED	20 60 40 20	FC 1F E3 FC	B1 C8 79 B1	5B ED B6 5B
0B	6A BE D4 6A	CB 46 8D CB	BE D9 67 BE	39 4B 72 39	4A DE 94 4A	4C D4 98 4C	58 E8 B0 58	CF 4A 85 CF
0C	D0 6B BB D0	EF 2A C5 EF	AA E5 4F AA	FB 16 ED FB	43 C5 86 43	4D D7 9A 4D	33 55 66 33	85 94 11 85
0D	45 CF 8A 45	F9 10 E9 F9	02 06 04 02	7F 81 FE 7F	50 F0 A0 50	3C 44 78 3C	9F BA 25 9F	A8 E3 4B A8
0E	51 F3 A2 51	A3 FE 5D A3	40 C0 80 40	8F 8A 05 8F	92 AD 3F 92	9D BC 21 9D	38 48 70 38	F5 04 F1 F5
0F	BC DF 63 BC	B6 C1 77 B6	DA 75 AF DA	21 63 42 21	10 30 20 10	FF 1A E5 FF	F3 0E FD F3	D2 6D BF D2
10	CD 4C 81 CD	0C 14 18 0C	13 35 26 13	EC 2F C3 EC	5F E1 BE 5F	97 A2 35 97	44 CC 88 44	17 39 2E 17
11	C4 57 93 C4	A7 F2 55 A7	7E 82 FC 7E	3D 47 7A 3D	64 AC C8 64	5D E7 BA 5D	19 2B 32 19	73 95 E6 73
12	60 A0 C0 60	81 98 19 81	4F D1 9E 4F	DC 7F A3 DC	22 66 44 22	2A 7E 54 2A	90 AB 3B 90	88 83 0B 88
13	46 CA 8C 46	EE 29 C7 EE	B8 D3 6B B8	14 3C 28 14	DE 79 A7 DE	5E E2 BC 5E	0B 1D 16 0B	DB 76 AD DB
14	E0 3B DB E0	32 56 64 32	3A 4E 74 3A	0A 1E 14 0A	49 DB 92 49	06 0A 0C 06	24 6C 48 24	5C E4 B8 5C
15	C2 5D 9F C2	D3 6E BD D3	AC EF 43 AC	62 A6 C4 62	91 A8 39 91	95 A4 31 95	E4 37 D3 E4	79 8B F2 79
16	E7 32 D5 E7	C8 43 8B C8	37 59 6E 37	6D B7 DA 6D	8D 8C 01 8D	D5 64 B1 D5	4E D2 9C 4E	A9 E0 49 A9
17	6C B4 D8 6C	56 FA AC 56	F4 07 F3 F4	EA 25 CF EA	65 AF CA 65	7A 8E F4 7A	AE E9 47 AE	08 18 10 08
18	BA D5 6F BA	78 88 F0 78	25 6F 4A 25	2E 72 5C 2E	1C 24 38 1C	A6 F1 57 A6	B4 C7 73 B4	C6 51 97 C6
19	E8 23 CB E8	DD 7C A1 DD	74 9C E8 74	1F 21 3E 1F	4B DD 96 4B	BD DC 61 BD	8B 86 0D 8B	8A 85 0F 8A
1A	70 90 E0 70	3E 42 7C 3E	B5 C4 71 B5	66 AA CC 66	48 D8 90 48	03 05 06 03	F6 01 F7 F6	0E 12 1C 0E
1B	61 A3 C2 61	35 5F 6A 35	57 F9 AE 57	B9 D0 69 B9	86 91 17 86	C1 58 99 C1	1D 27 3A 1D	9E B9 27 9E
1C	E1 38 D9 E1	F8 13 EB F8	98 B3 2B 98	11 33 22 11	69 BB D2 69	D9 70 A9 D9	8E 89 07 8E	94 A7 33 94
1D	9B B6 2D 9B	1E 22 3C 1E	87 92 15 87	E9 20 C9 E9	CE 49 87 CE	55 FF AA 55	28 78 50 28	DF 7A A5 DF
1E	8C 8F 03 8C	A1 F8 59 A1	89 80 09 89	0D 17 1A 0D	BF DA 65 BF	E6 31 D7 E6	42 C6 84 42	68 B8 D0 68
1F	41 C3 82 41	99 B0 29 99	2D 77 5A 2D	0F 11 1E 0F	B0 CB 7B B0	54 FC A8 54	BB D6 6D BB	16 3A 2C 16

Table A.3: Sbox $\mathbf{T}_2$

	0	1	2	3	4	5	6	7
00	63 63 A5 C6	7C 7C 84 F8	77 77 99 EE	7B 7B 8D F6	F2 F2 0D FF	6B 6B BD D6	6F 6F B1 DE	C5 C5 54 91
01	30 30 50 60	01 01 03 02	67 67 A9 CE	2B 2B 7D 56	FE FE 19 E7	D7 D7 62 B5	AB AB E6 4D	76 76 9A EC
02	CA CA 45 8F	82 82 9D 1F	C9 C9 40 89	7D 7D 87 FA	FA FA 15 EF	59 59 EB B2	47 47 C9 8E	F0 F0 0B FB
03	AD AD EC 41	D4 D4 67 B3	A2 A2 FD 5F	AF AF EA 45	9C 9C BF 23	A4 A4 F7 53	72 72 96 E4	C0 C0 5B 9B
04	B7 B7 C2 75	FD FD 1C E1	93 93 AE 3D	26 26 6A 4C	36 36 5A 6C	3F 3F 41 7E	F7 F7 02 F5	CC CC 4F 83
05	34 34 5C 68	A5 A5 F4 51	E5 E5 34 D1	F1 F1 08 F9	71 71 93 E2	D8 D8 73 AB	31 31 53 62	15 15 3F 2A
06	04 04 0C 08	C7 C7 52 95	23 23 65 46	C3 C3 5E 9D	18 18 28 30	96 96 A1 37	05 05 0F 0A	9A 9A B5 2F
07	07 07 09 0E	12 12 36 24	80 80 9B 1B	E2 E2 3D DF	EB EB 26 CD	27 27 69 4E	B2 B2 CD 7F	75 75 9F EA
08	09 09 1B 12	83 83 9E 1D	2C 2C 74 58	1A 1A 2E 34	1B 1B 2D 36	6E 6E B2 DC	5A 5A EE B4	A0 A0 FB 5B
09	52 52 F6 A4	3B 3B 4D 76	D6 D6 61 B7	B3 B3 CE 7D	29 29 7B 52	E3 E3 3E DD	2F 2F 71 5E	84 84 97 13
0A	53 53 F5 A6	D1 D1 68 B9	00 00 00 00	ED ED 2C C1	20 20 60 40	FC FC 1F E3	B1 B1 C8 79	5B 5B ED B6
0B	6A 6A BE D4	CB CB 46 8D	BE BE D9 67	39 39 4B 72	4A 4A DE 94	4C 4C D4 98	58 58 E8 B0	CF CF 4A 85
0C	D0 D0 6B BB	EF EF 2A C5	AA AA E5 4F	FB FB 16 ED	43 43 C5 86	4D 4D D7 9A	33 33 55 66	85 85 94 11
0D	45 45 CF 8A	F9 F9 10 E9	02 02 06 04	7F 7F 81 FE	50 50 F0 A0	3C 3C 44 78	9F 9F BA 25	A8 A8 E3 4B
0E	51 51 F3 A2	A3 A3 FE 5D	40 40 C0 80	8F 8F 8A 05	92 92 AD 3F	9D 9D BC 21	38 38 48 70	F5 F5 04 F1
0F	BC BC DF 63	B6 B6 C1 77	DA DA 75 AF	21 21 63 42	10 10 30 20	FF FF 1A E5	F3 F3 0E FD	D2 D2 6D BF
10	CD CD 4C 81	0C 0C 14 18	13 13 35 26	EC EC 2F C3	5F 5F E1 BE	97 97 A2 35	44 44 CC 88	17 17 39 2E
11	C4 C4 57 93	A7 A7 F2 55	7E 7E 82 FC	3D 3D 47 7A	64 64 AC C8	5D 5D E7 BA	19 19 2B 32	73 73 95 E6
12	60 60 A0 C0	81 81 98 19	4F 4F D1 9E	DC DC 7F A3	22 22 66 44	2A 2A 7E 54	90 90 AB 3B	88 88 83 0B
13	46 46 CA 8C	EE EE 29 C7	B8 B8 D3 6B	14 14 3C 28	DE DE 79 A7	5E 5E E2 BC	0B 0B 1D 16	DB DB 76 AD
14	E0 E0 3B DB	32 32 56 64	3A 3A 4E 74	0A 0A 1E 14	49 49 DB 92	06 06 0A 0C	24 24 6C 48	5C 5C E4 B8
15	C2 C2 5D 9F	D3 D3 6E BD	AC AC EF 43	62 62 A6 C4	91 91 A8 39	95 95 A4 31	E4 E4 37 D3	79 79 8B F2
16	E7 E7 32 D5	C8 C8 43 8B	37 37 59 6E	6D 6D B7 DA	8D 8D 8C 01	D5 D5 64 B1	4E 4E D2 9C	A9 A9 E0 49
17	6C 6C B4 D8	56 56 FA AC	F4 F4 07 F3	EA EA 25 CF	65 65 AF CA	7A 7A 8E F4	AE AE E9 47	08 08 18 10
18	BA BA D5 6F	78 78 88 F0	25 25 6F 4A	2E 2E 72 5C	1C 1C 24 38	A6 A6 F1 57	B4 B4 C7 73	C6 C6 51 97
19	E8 E8 23 CB	DD DD 7C A1	74 74 9C E8	1F 1F 21 3E	4B 4B DD 96	BD BD DC 61	8B 8B 86 0D	8A 8A 85 0F
1A	70 70 90 E0	3E 3E 42 7C	B5 B5 C4 71	66 66 AA CC	48 48 D8 90	03 03 05 06	F6 F6 01 F7	0E 0E 12 1C
1B	61 61 A3 C2	35 35 5F 6A	57 57 F9 AE	B9 B9 D0 69	86 86 91 17	C1 C1 58 99	1D 1D 27 3A	9E 9E B9 27
1C	E1 E1 38 D9	F8 F8 13 EB	98 98 B3 2B	11 11 33 22	69 69 BB D2	D9 D9 70 A9	8E 8E 89 07	94 94 A7 33
1D	9B 9B B6 2D	1E 1E 22 3C	87 87 92 15	E9 E9 20 C9	CE CE 49 87	55 55 FF AA	28 28 78 50	DF DF 7A A5
1E	8C 8C 8F 03	A1 A1 F8 59	89 89 80 09	0D 0D 17 1A	BF BF DA 65	E6 E6 31 D7	42 42 C6 84	68 68 B8 D0
1F	41 41 C3 82	99 99 B0 29	2D 2D 77 5A	0F 0F 11 1E	B0 B0 CB 7B	54 54 FC A8	BB BB D6 6D	16 16 3A 2C

Table A.4: Sbox $\mathbf{T}_3$

	0	1	2	3	4	5	6	7
00	63 63 63 63	7C 7C 7C 7C	77 77 77 77	7B 7B 7B 7B	F2 F2 F2 F2	6B 6B 6B 6B	6F 6F 6F 6F	C5 C5 C5 C5
01	30 30 30 30	01 01 01 01	67 67 67 67	2B 2B 2B 2B	FE FE FE FE	D7 D7 D7 D7	AB AB AB AB	76 76 76 76
02	CA CA CA CA	82 82 82 82	C9 C9 C9 C9	7D 7D 7D 7D	FA FA FA FA	59 59 59 59	47 47 47 47	F0 F0 F0 F0
03	AD AD AD AD	D4 D4 D4 D4	A2 A2 A2 A2	AF AF AF AF	9C 9C 9C 9C	A4 A4 A4 A4	72 72 72 72	C0 C0 C0 C0
04	B7 B7 B7 B7	FD FD FD FD	93 93 93 93	26 26 26 26	36 36 36 36	3F 3F 3F 3F	F7 F7 F7 F7	CC CC CC CC
05	34 34 34 34	A5 A5 A5 A5	E5 E5 E5 E5	F1 F1 F1 F1	71 71 71 71	D8 D8 D8 D8	31 31 31 31	15 15 15 15
06	04 04 04 04	C7 C7 C7 C7	23 23 23 23	C3 C3 C3 C3	18 18 18 18	96 96 96 96	05 05 05 05	9A 9A 9A 9A
07	07 07 07 07	12 12 12 12	80 80 80 80	E2 E2 E2 E2	EB EB EB EB	27 27 27 27	B2 B2 B2 B2	75 75 75 75
08	09 09 09 09	83 83 83 83	2C 2C 2C 2C	1A 1A 1A 1A	1B 1B 1B 1B	6E 6E 6E 6E	5A 5A 5A 5A	A0 A0 A0 A0
09	52 52 52 52	3B 3B 3B 3B	D6 D6 D6 D6	B3 B3 B3 B3	29 29 29 29	E3 E3 E3 E3	2F 2F 2F 2F	84 84 84 84
0A	53 53 53 53	D1 D1 D1 D1	00 00 00 00	ED ED ED ED	20 20 20 20	FC FC FC FC	B1 B1 B1 B1	5B 5B 5B 5B
0B	6A 6A 6A 6A	CB CB CB CB	BE BE BE BE	39 39 39 39	4A 4A 4A 4A	4C 4C 4C 4C	58 58 58 58	CF CF CF CF
0C	D0 D0 D0 D0	EF EF EF EF	AA AA AA AA	FB FB FB FB	43 43 43 43	4D 4D 4D 4D	33 33 33 33	85 85 85 85
0D	45 45 45 45	F9 F9 F9 F9	02 02 02 02	7F 7F 7F 7F	50 50 50 50	3C 3C 3C 3C	9F 9F 9F 9F	A8 A8 A8 A8
0E	51 51 51 51	A3 A3 A3 A3	40 40 40 40	8F 8F 8F 8F	92 92 92 92	9D 9D 9D 9D	38 38 38 38	F5 F5 F5 F5
0F	BC BC BC BC	B6 B6 B6 B6	DA DA DA DA	21 21 21 21	10 10 10 10	FF FF FF FF	F3 F3 F3 F3	D2 D2 D2 D2
10	CD CD CD CD	0C 0C 0C 0C	13 13 13 13	EC EC EC EC	5F 5F 5F 5F	97 97 97 97	44 44 44 44	17 17 17 17
11	C4 C4 C4 C4	A7 A7 A7 A7	7E 7E 7E 7E	3D 3D 3D 3D	64 64 64 64	5D 5D 5D 5D	19 19 19 19	73 73 73 73
12	60 60 60 60	81 81 81 81	4F 4F 4F 4F	DC DC DC DC	22 22 22 22	2A 2A 2A 2A	90 90 90 90	88 88 88 88
13	46 46 46 46	EE EE EE EE	B8 B8 B8 B8	14 14 14 14	DE DE DE DE	5E 5E 5E 5E	0B 0B 0B 0B	DB DB DB DB
14	E0 E0 E0 E0	32 32 32 32	3A 3A 3A 3A	0A 0A 0A 0A	49 49 49 49	06 06 06 06	24 24 24 24	5C 5C 5C 5C
15	C2 C2 C2 C2	D3 D3 D3 D3	AC AC AC AC	62 62 62 62	91 91 91 91	95 95 95 95	E4 E4 E4 E4	79 79 79 79
16	E7 E7 E7 E7	C8 C8 C8 C8	37 37 37 37	6D 6D 6D 6D	8D 8D 8D 8D	D5 D5 D5 D5	4E 4E 4E 4E	A9 A9 A9 A9
17	6C 6C 6C 6C	56 56 56 56	F4 F4 F4 F4	EA EA EA EA	65 65 65 65	7A 7A 7A 7A	AE AE AE AE	08 08 08 08
18	BA BA BA BA	78 78 78 78	25 25 25 25	2E 2E 2E 2E	1C 1C 1C 1C	A6 A6 A6 A6	B4 B4 B4 B4	C6 C6 C6 C6
19	E8 E8 E8 E8	DD DD DD DD	74 74 74 74	1F 1F 1F 1F	4B 4B 4B 4B	BD BD BD BD	8B 8B 8B 8B	8A 8A 8A 8A
1A	70 70 70 70	3E 3E 3E 3E	B5 B5 B5 B5	66 66 66 66	48 48 48 48	03 03 03 03	F6 F6 F6 F6	0E 0E 0E 0E
1B	61 61 61 61	35 35 35 35	57 57 57 57	B9 B9 B9 B9	86 86 86 86	C1 C1 C1 C1	1D 1D 1D 1D	9E 9E 9E 9E
1C	E1 E1 E1 E1	F8 F8 F8 F8	98 98 98 98	11 11 11 11	69 69 69 69	D9 D9 D9 D9	8E 8E 8E 8E	94 94 94 94
1D	9B 9B 9B 9B	1E 1E 1E 1E	87 87 87 87	E9 E9 E9 E9	CE CE CE CE	55 55 55 55	28 28 28 28	DF DF DF DF
1E	8C 8C 8C 8C	A1 A1 A1 A1	89 89 89 89	0D 0D 0D 0D	BF BF BF BF	E6 E6 E6 E6	42 42 42 42	68 68 68 68
1F	41 41 41 41	99 99 99 99	2D 2D 2D 2D	0F 0F 0F 0F	B0 B0 B0 B0	54 54 54 54	BB BB BB BB	16 16 16 16

Table A.5: Sbox $\mathbf{T}_4$

Appendix B

Decompositions of the AES Sbox

In the sequel, the standard AES sbox is decomposed into smaller number of sboxes as described in Section 6.6 on page 97. For each decomposition the function to compute $\mathbf{S}[x]$ given x is shown.

The standard AES Sbox The standard sbox is an efficient realization of the mapping

$$\begin{array}{rcl} \{0,1\}^8 & \rightarrow & \{0,1\}^8 \\ x & \mapsto & \mathbf{S}[x]. \end{array}$$

$$\begin{bmatrix}
63 & 7C & 77 & 7B & F2 & 6B & 6F & C5 & 30 & 01 & 67 & 2B & FE & D7 & AB & 76 \\
CA & 82 & C9 & 7D & FA & 59 & 47 & F0 & AD & D4 & A2 & AF & 9C & A4 & 72 & C0 \\
B7 & FD & 93 & 26 & 36 & 3F & F7 & CC & 34 & A5 & E5 & F1 & 71 & D8 & 31 & 15 \\
04 & C7 & 23 & C3 & 18 & 96 & 05 & 9A & 07 & 12 & 80 & E2 & EB & 27 & B2 & 75 \\
09 & 83 & 2C & 1A & 1B & 6E & 5A & A0 & 52 & 3B & D6 & B3 & 29 & E3 & 2F & 84 \\
53 & D1 & 00 & ED & 20 & FC & B1 & 5B & 6A & CB & BE & 39 & 4A & 4C & 58 & CF \\
D0 & EF & AA & FB & 43 & 4D & 33 & 85 & 45 & F9 & 02 & 7F & 50 & 3C & 9F & A8 \\
51 & A3 & 40 & 8F & 92 & 9D & 38 & F5 & BC & B6 & DA & 21 & 10 & FF & F3 & D2 \\
CD & 0C & 13 & EC & 5F & 97 & 44 & 17 & C4 & A7 & 7E & 3D & 64 & 5D & 19 & 73 \\
60 & 81 & 4F & DC & 22 & 2A & 90 & 88 & 46 & EE & B8 & 14 & DE & 5E & 0B & DB \\
E0 & 32 & 3A & 0A & 49 & 06 & 24 & 5C & C2 & D3 & AC & 62 & 91 & 95 & E4 & 79 \\
E7 & C8 & 37 & 6D & 8D & D5 & 4E & A9 & 6C & 56 & F4 & EA & 65 & 7A & AE & 08 \\
BA & 78 & 25 & 2E & 1C & A6 & B4 & C6 & E8 & DD & 74 & 1F & 4B & BD & 8B & 8A \\
70 & 3E & B5 & 66 & 48 & 03 & F6 & 0E & 61 & 35 & 57 & B9 & 86 & C1 & 1D & 9E \\
E1 & F8 & 98 & 11 & 69 & D9 & 8E & 94 & 9B & 1E & 87 & E9 & CE & 55 & 28 & DF \\
8C & A1 & 89 & 0D & BF & E6 & 42 & 68 & 41 & 99 & 2D & 0F & B0 & 54 & BB & 16
\end{bmatrix}$$

Table B.1: The standard sbox $\mathbf{S}$

Decomposition of the sbox S into 2 smaller sboxes

The standard sbox $\mathbf{S}$ is splitted into 2 smaller sboxes $\mathbf{S}_0^{(2)}$ and $\mathbf{S}_1^{(2)}$ each mapping from $\{0,1\}^8$ to $\{0,1\}^4$. The application of the sbox is then realized as

$$\begin{array}{rcl} \{0,1\}^8 & \rightarrow & \{0,1\}^4 \times \{0,1\}^4 \\ x & \mapsto & 16 \cdot \mathbf{S}_1^{(2)}[x] \oplus \mathbf{S}_0^{(2)}[x] \end{array}$$

$$\mathbf{S}_0^{(2)}: \begin{bmatrix}
3 & C & 7 & B & 2 & B & F & 5 & 0 & 1 & 7 & B & E & 7 & B & 6 \\
A & 2 & 9 & D & A & 9 & 7 & 0 & D & 4 & 2 & F & C & 4 & 2 & 0 \\
7 & D & 3 & 6 & 6 & F & 7 & C & 4 & 5 & 5 & 1 & 1 & 8 & 1 & 5 \\
4 & 7 & 3 & 3 & 8 & 6 & 5 & A & 7 & 2 & 0 & 2 & B & 7 & 2 & 5 \\
9 & 3 & C & A & B & E & A & 0 & 2 & B & 6 & 3 & 9 & 3 & F & 4 \\
3 & 1 & 0 & D & 0 & C & 1 & B & A & B & E & 9 & A & C & 8 & F \\
0 & F & A & B & 3 & D & 3 & 5 & 5 & 9 & 2 & F & 0 & C & F & 8 \\
1 & 3 & 0 & F & 2 & D & 8 & 5 & C & 6 & A & 1 & 0 & F & 3 & 2 \\
D & C & 3 & C & F & 7 & 4 & 7 & 4 & 7 & E & D & 4 & D & 9 & 3 \\
0 & 1 & F & C & 2 & A & 0 & 8 & 6 & E & 8 & 4 & E & E & B & B \\
0 & 2 & A & A & 9 & 6 & 4 & C & 2 & 3 & C & 2 & 1 & 5 & 4 & 9 \\
7 & 8 & 7 & D & D & 5 & E & 9 & C & 6 & 4 & A & 5 & A & E & 8 \\
A & 8 & 5 & E & C & 6 & 4 & 6 & 8 & D & 4 & F & B & D & B & A \\
0 & E & 5 & 6 & 8 & 3 & 6 & E & 1 & 5 & 7 & 9 & 6 & 1 & D & E \\
1 & 8 & 8 & 1 & 9 & 9 & E & 4 & B & E & 7 & 9 & E & 5 & 8 & F \\
C & 1 & 9 & D & F & 6 & 2 & 8 & 1 & 9 & D & F & 0 & 4 & B & 6
\end{bmatrix}$$

$$\mathbf{S}_1^{(2)}: \begin{bmatrix}
6 & 7 & 7 & 7 & F & 6 & 6 & C & 3 & 0 & 6 & 2 & F & D & A & 7 \\
C & 8 & C & 7 & F & 5 & 4 & F & A & D & A & A & 9 & A & 7 & C \\
B & F & 9 & 2 & 3 & 3 & F & C & 3 & A & E & F & 7 & D & 3 & 1 \\
0 & C & 2 & C & 1 & 9 & 0 & 9 & 0 & 1 & 8 & E & E & 2 & B & 7 \\
0 & 8 & 2 & 1 & 1 & 6 & 5 & A & 5 & 3 & D & B & 2 & E & 2 & 8 \\
5 & D & 0 & E & 2 & F & B & 5 & 6 & C & B & 3 & 4 & 4 & 5 & C \\
D & E & A & F & 4 & 4 & 3 & 8 & 4 & F & 0 & 7 & 5 & 3 & 9 & A \\
5 & A & 4 & 8 & 9 & 9 & 3 & F & B & B & D & 2 & 1 & F & F & D \\
C & 0 & 1 & E & 5 & 9 & 4 & 1 & C & A & 7 & 3 & 6 & 5 & 1 & 7 \\
6 & 8 & 4 & D & 2 & 2 & 9 & 8 & 4 & E & B & 1 & D & 5 & 0 & D \\
E & 3 & 3 & 0 & 4 & 0 & 2 & 5 & C & D & A & 6 & 9 & 9 & E & 7 \\
E & C & 3 & 6 & 8 & D & 4 & A & 6 & 5 & F & E & 6 & 7 & A & 0 \\
B & 7 & 2 & 2 & 1 & A & B & C & E & D & 7 & 1 & 4 & B & 8 & 8 \\
7 & 3 & B & 6 & 4 & 0 & F & 0 & 6 & 3 & 5 & B & 8 & C & 1 & 9 \\
E & F & 9 & 1 & 6 & D & 8 & 9 & 9 & 1 & 8 & E & C & 5 & 2 & D \\
8 & A & 8 & 0 & B & E & 4 & 6 & 4 & 9 & 2 & 0 & B & 5 & B & 1
\end{bmatrix}$$

Decomposition of the sbox S into 4 smaller sboxes

The standard sbox $\mathbf{S}$ is splitted into 4 smaller sboxes $\mathbf{S}_0^{(4)}, \ldots, \mathbf{S}_3^{(4)}$ each mapping from $\{0,1\}^8$ to $\{0,1\}^2$. The application of the sbox is then realized as

$$x \mapsto 64 \cdot \mathbf{S}_3^{(4)}[x] \oplus 16 \cdot \mathbf{S}_2^{(4)}[x] \oplus 4 \cdot \mathbf{S}_1^{(4)}[x] \oplus \mathbf{S}_0^{(4)}[x]$$

$$\mathbf{S}_0^{(4)}: \begin{bmatrix}
3 & 0 & 3 & 3 & 2 & 3 & 3 & 1 & 0 & 1 & 3 & 3 & 2 & 3 & 3 & 2 \\
2 & 2 & 1 & 1 & 2 & 1 & 3 & 0 & 1 & 0 & 2 & 3 & 0 & 0 & 2 & 0 \\
3 & 1 & 3 & 2 & 2 & 3 & 3 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 3 & 3 & 3 & 0 & 2 & 1 & 2 & 3 & 2 & 0 & 2 & 3 & 3 & 2 & 1 \\
1 & 3 & 0 & 2 & 3 & 2 & 2 & 0 & 2 & 3 & 2 & 3 & 1 & 3 & 3 & 0 \\
3 & 1 & 0 & 1 & 0 & 0 & 1 & 3 & 2 & 3 & 2 & 1 & 2 & 0 & 0 & 3 \\
0 & 3 & 2 & 3 & 3 & 1 & 3 & 1 & 1 & 1 & 2 & 3 & 0 & 0 & 3 & 0 \\
1 & 3 & 0 & 3 & 2 & 1 & 0 & 1 & 0 & 2 & 2 & 1 & 0 & 3 & 3 & 2 \\
1 & 0 & 3 & 0 & 3 & 3 & 0 & 3 & 0 & 3 & 2 & 1 & 0 & 1 & 1 & 3 \\
0 & 1 & 3 & 0 & 2 & 2 & 0 & 0 & 2 & 2 & 0 & 0 & 2 & 2 & 3 & 3 \\
0 & 2 & 2 & 2 & 1 & 2 & 0 & 0 & 2 & 3 & 0 & 2 & 1 & 1 & 0 & 1 \\
3 & 0 & 3 & 1 & 1 & 1 & 2 & 1 & 0 & 2 & 0 & 2 & 1 & 2 & 2 & 0 \\
2 & 0 & 1 & 2 & 0 & 2 & 0 & 2 & 0 & 1 & 0 & 3 & 3 & 1 & 3 & 2 \\
0 & 2 & 1 & 2 & 0 & 3 & 2 & 2 & 1 & 1 & 3 & 1 & 2 & 1 & 1 & 2 \\
1 & 0 & 0 & 1 & 1 & 1 & 2 & 0 & 3 & 2 & 3 & 1 & 2 & 1 & 0 & 3 \\
0 & 1 & 1 & 1 & 3 & 2 & 2 & 0 & 1 & 1 & 1 & 3 & 0 & 0 & 3 & 2
\end{bmatrix}$$

$$\mathbf{S}_1^{(4)}: \begin{bmatrix}
0 & 3 & 1 & 2 & 0 & 2 & 3 & 1 & 0 & 0 & 1 & 2 & 3 & 1 & 2 & 1 \\
2 & 0 & 2 & 3 & 2 & 2 & 1 & 0 & 3 & 1 & 0 & 3 & 3 & 1 & 0 & 0 \\
1 & 3 & 0 & 1 & 1 & 3 & 1 & 3 & 1 & 1 & 1 & 0 & 0 & 2 & 0 & 1 \\
1 & 1 & 0 & 0 & 2 & 1 & 1 & 2 & 1 & 0 & 0 & 0 & 2 & 1 & 0 & 1 \\
2 & 0 & 3 & 2 & 2 & 3 & 2 & 0 & 0 & 2 & 1 & 0 & 2 & 0 & 3 & 1 \\
0 & 0 & 0 & 3 & 0 & 3 & 0 & 2 & 2 & 2 & 3 & 2 & 2 & 3 & 2 & 3 \\
0 & 3 & 2 & 2 & 0 & 3 & 0 & 1 & 1 & 2 & 0 & 3 & 0 & 3 & 3 & 2 \\
0 & 0 & 0 & 3 & 0 & 3 & 2 & 1 & 3 & 1 & 2 & 0 & 0 & 3 & 0 & 0 \\
3 & 3 & 0 & 3 & 3 & 1 & 1 & 1 & 1 & 1 & 3 & 3 & 1 & 3 & 2 & 0 \\
0 & 0 & 3 & 3 & 0 & 2 & 0 & 2 & 1 & 3 & 2 & 1 & 3 & 3 & 2 & 2 \\
0 & 0 & 2 & 2 & 2 & 1 & 1 & 3 & 0 & 0 & 3 & 0 & 0 & 1 & 1 & 2 \\
1 & 2 & 1 & 3 & 3 & 1 & 3 & 2 & 3 & 1 & 1 & 2 & 1 & 2 & 3 & 2 \\
2 & 2 & 1 & 3 & 3 & 1 & 1 & 1 & 2 & 3 & 1 & 3 & 2 & 3 & 2 & 2 \\
0 & 3 & 1 & 1 & 2 & 0 & 1 & 3 & 0 & 1 & 1 & 2 & 1 & 0 & 3 & 3 \\
0 & 2 & 2 & 0 & 2 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \\
3 & 0 & 2 & 3 & 3 & 1 & 0 & 2 & 0 & 2 & 3 & 3 & 0 & 1 & 2 & 1
\end{bmatrix}$$

$$\mathbf{S}_2^{(4)}:\ \begin{bmatrix}
2&3&3&3&3&2&2&0&3&0&2&2&3&1&2&3\\
0&0&0&3&3&1&0&3&2&1&2&2&1&2&3&0\\
3&3&1&2&3&3&3&0&3&2&2&3&3&1&3&1\\
0&0&2&0&1&1&0&1&0&1&0&2&2&2&3&3\\
0&0&2&1&1&2&1&2&1&3&1&3&2&2&2&0\\
1&1&0&2&2&3&3&1&2&0&3&3&0&0&1&0\\
1&2&2&3&0&0&3&0&0&3&0&3&1&3&1&2\\
1&2&0&0&1&1&3&3&3&3&1&2&1&3&3&1\\
0&0&1&2&1&1&0&1&0&2&3&3&2&1&1&3\\
2&0&0&1&2&2&1&0&0&2&3&1&1&1&0&1\\
2&3&3&0&0&0&2&1&0&1&2&2&1&1&2&3\\
2&0&3&2&0&1&0&2&2&1&3&2&2&3&2&0\\
3&3&2&2&1&2&3&0&2&1&3&1&0&3&0&0\\
3&3&3&2&0&0&3&0&2&3&1&3&0&0&1&1\\
2&3&1&1&2&1&0&1&1&1&0&2&0&1&2&1\\
0&2&0&0&3&2&0&2&0&1&2&0&3&1&3&1
\end{bmatrix}$$

$$\mathbf{S}_3^{(4)}:\ \begin{bmatrix}
1&1&1&1&3&1&1&3&0&0&1&0&3&3&2&1\\
3&2&3&1&3&1&1&3&2&3&2&2&2&2&1&3\\
2&3&2&0&0&0&3&3&0&2&3&3&1&3&0&0\\
0&3&0&3&0&2&0&2&0&0&2&3&3&0&2&1\\
0&2&0&0&0&1&1&2&1&0&3&2&0&3&0&2\\
1&3&0&3&0&3&2&1&1&3&2&0&1&1&1&3\\
3&3&2&3&1&1&0&2&1&3&0&1&1&0&2&2\\
1&2&1&2&2&2&0&3&2&2&3&0&0&3&3&3\\
3&0&0&3&1&2&1&0&3&2&1&0&1&1&0&1\\
1&2&1&3&0&0&2&2&1&3&2&0&3&1&0&3\\
3&0&0&0&1&0&0&1&3&3&2&1&2&2&3&1\\
3&3&0&1&2&3&1&2&1&1&3&3&1&1&2&0\\
2&1&0&0&0&2&2&3&3&3&1&0&1&2&2&2\\
1&0&2&1&1&0&3&0&1&0&1&2&2&3&0&2\\
3&3&2&0&1&3&2&2&2&0&2&3&3&1&0&3\\
2&2&2&0&2&3&1&1&1&2&0&0&2&1&2&0
\end{bmatrix}$$

Decomposition of the sbox S into 8 smaller sboxes

The standard sbox $\mathbf{S}$ is splitted into 8 smaller sboxes $\mathbf{S}_0^{(8)}, \ldots, \mathbf{S}_7^{(8)}$ each mapping from $\{0,1\}^8$ to $\{0,1\}^1$. The application of the sbox is then realized as

$$\{0,1\}^8 \to \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\}$$

$$x \mapsto 128 \cdot \mathbf{S}_7^{(8)}[x] \oplus 64 \cdot \mathbf{S}_6^{(8)}[x] \oplus 32 \cdot \mathbf{S}_5^{(8)}[x] \oplus 16 \cdot \mathbf{S}_4^{(8)}[x] \oplus 8 \cdot \mathbf{S}_3^{(8)}[x] \oplus 4 \cdot \mathbf{S}_2^{(8)}[x] \oplus 2 \cdot \mathbf{S}_1^{(8)}[x] \oplus \mathbf{S}_0^{(8)}[x]$$

$$\mathbf{S}_0^{(8)}: \begin{bmatrix}
1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}$$

$$\mathbf{S}_1^{(8)}: \begin{bmatrix}
1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1
\end{bmatrix}$$

$$
\mathbf{S}_2^{(8)}: \left[\begin{array}{cccccccccccccccc}
0&1&1&0&0&0&1&1&0&0&1&0&1&1&0&1\\
0&0&0&1&0&0&1&0&1&1&0&1&1&1&0&0\\
1&1&0&1&1&1&1&1&1&1&1&0&0&0&0&1\\
1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&1\\
0&0&1&0&0&1&0&0&0&0&1&0&0&0&1&1\\
0&0&0&1&0&1&0&0&0&0&1&0&0&1&0&1\\
0&1&0&0&0&1&0&1&1&0&0&1&0&1&1&0\\
0&0&0&1&0&1&0&1&1&1&0&0&0&1&0&0\\
1&1&0&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&0&0&0&0&1&1&0&1&1&1&0&0\\
0&0&0&0&0&1&1&1&0&0&1&0&0&1&1&0\\
1&0&1&1&1&1&1&0&1&1&1&0&1&0&1&0\\
0&0&1&1&1&1&1&1&0&1&1&1&0&1&0&0\\
0&1&1&1&0&0&1&1&0&1&1&0&1&0&1&1\\
0&0&0&0&0&0&1&1&0&1&1&0&1&1&0&1\\
1&0&0&1&1&1&0&0&0&0&1&1&0&1&0&1
\end{array}\right]
$$

$$
\mathbf{S}_3^{(8)}: \left[\begin{array}{cccccccccccccccc}
0&1&0&1&0&1&1&0&0&0&0&1&1&0&1&0\\
1&0&1&1&1&1&0&0&1&0&0&1&1&0&0&0\\
0&1&0&0&0&1&0&1&0&0&0&0&0&1&0&0\\
0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&0\\
1&0&1&1&1&1&1&0&0&1&0&0&1&0&1&0\\
0&0&0&1&0&1&0&1&1&1&1&1&1&1&1&1\\
0&1&1&1&0&1&0&0&0&1&0&1&0&1&1&1\\
0&0&0&1&0&1&1&0&1&0&1&0&0&1&0&0\\
1&1&0&1&1&0&0&0&0&0&1&1&0&1&1&0\\
0&0&1&1&0&1&0&1&0&1&1&0&1&1&1&1\\
0&0&1&1&1&0&0&1&0&0&1&0&0&0&0&1\\
0&1&0&1&1&0&1&1&1&0&0&1&0&1&1&1\\
1&1&0&1&1&0&0&0&1&1&0&1&1&1&1&1\\
0&1&0&0&1&0&0&1&0&0&0&1&0&0&1&1\\
0&1&1&0&1&1&1&0&1&1&0&1&1&0&1&1\\
1&0&1&1&1&0&0&1&0&1&1&1&0&0&1&0
\end{array}\right]
$$

$$\mathbf{S}_4^{(8)}\colon \quad \begin{bmatrix}
0&1&1&1&1&0&0&0&1&0&0&0&1&1&0&1\\
0&0&0&1&1&1&0&1&0&1&0&0&1&0&1&0\\
1&1&1&0&1&1&1&0&1&0&0&1&1&1&1&1\\
0&0&0&0&1&1&0&1&0&1&0&0&0&0&1&1\\
0&0&0&1&1&0&1&0&1&1&1&1&0&0&0&0\\
1&1&0&0&0&1&1&1&0&0&1&1&0&0&1&0\\
1&0&0&1&0&0&1&0&0&1&0&1&1&1&1&0\\
1&0&0&0&1&1&1&1&1&1&1&0&1&1&1&1\\
0&0&1&0&1&1&0&1&0&0&1&1&0&1&1&1\\
0&0&0&1&0&0&1&0&0&0&1&1&1&1&0&1\\
0&1&1&0&0&0&0&1&0&1&0&0&1&1&0&1\\
0&0&1&0&0&1&0&0&0&1&1&0&0&1&0&0\\
1&1&0&0&1&0&1&0&0&1&1&1&0&1&0&0\\
1&1&1&0&0&0&1&0&0&1&1&1&0&0&1&1\\
0&1&1&1&0&1&0&1&1&1&0&0&0&1&0&1\\
0&0&0&0&1&0&0&0&0&1&0&0&1&1&1&1
\end{bmatrix}$$

$$\mathbf{S}_5^{(8)}\colon \quad \begin{bmatrix}
1&1&1&1&1&1&1&0&1&0&1&1&1&0&1&1\\
0&0&0&1&1&0&0&1&1&0&1&1&0&1&1&0\\
1&1&0&1&1&1&1&0&1&1&1&1&1&0&1&0\\
0&0&1&0&0&0&0&0&0&0&0&1&1&1&1&1\\
0&0&1&0&0&1&0&1&0&1&0&1&1&1&1&0\\
0&0&0&1&1&1&1&0&1&0&1&1&0&0&0&0\\
0&1&1&1&0&0&1&0&0&1&0&1&0&1&0&1\\
0&1&0&0&0&0&1&1&1&1&0&1&0&1&1&0\\
0&0&0&1&0&0&0&0&0&1&1&1&1&0&0&1\\
1&0&0&0&1&1&0&0&0&1&1&0&0&0&0&0\\
1&1&1&0&0&0&1&0&0&0&1&1&0&0&1&1\\
1&0&1&1&0&0&0&1&1&0&1&1&1&1&1&0\\
1&1&1&1&0&1&1&0&1&0&1&0&0&1&0&0\\
1&1&1&1&0&0&1&0&1&1&0&1&0&0&0&0\\
1&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0\\
0&1&0&0&1&1&0&1&0&0&1&0&1&0&1&0
\end{bmatrix}$$

$$
\mathbf{S}_6^{(8)}\colon \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}
$$

$$
\mathbf{S}_7^{(8)}\colon \begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}
$$

List of Tables

4.1 Computation of $(u^{254} \oplus r_{1,13})$ using repeated squaring . . . 35

4.2 Computation of $(u^{254} \oplus r_1)$ using repeated squaring (simplified version) . . . 38

4.3 Hardware costs of different inversion circuits . . . 40

5.1 All possible differences of p_0,p'_0 . . . 66

5.2 Overview over the fault based collision attacks . . . 69

6.1 The memory hierarchy . . . 74

6.2 Comparing properties of different CBAs . . . 85

6.3 Experimental environment . . . 92

6.4 The resistance of the standard implementation . . . 93

6.5 The resistance of the fast implementation . . . 95

6.6 The resistance of the fast implementation using only $\mathbf{T}_0$ . . . 96

6.7 The resistance of small-2 . . . 98

6.8 Timings for small-2, small-4 and small-8 applied on the last round of AES . . . 98

6.9 Information Leakage, Resistance and Efficiency of AES implementations . . . 99

6.10 Simplified Comparison of Implementations . . . 100

A.1 Sbox $\mathbf{T}_0$ . . . 109

A.2 Sbox $\mathbf{T}_1$ . . . 110

A.3 Sbox $\mathbf{T}_2$ . . . 111

A.4 Sbox $\mathbf{T}_3$ . . . 112

A.5 Sbox $\mathbf{T}_4$ . . . 113

B.1 The standard sbox $\mathbf{S}$ . . . 115

List of Figures

2.1 Mapping the plaintext p into a state 9
2.2 The `SubBytes` transformation 10
2.3 The `ShiftRows` transformation 11
2.4 The `MixColumns` transformation 11
2.5 The `AddRoundKey` transformation 12

3.1 Black box model of classical cryptography 19
3.2 Extended black box model that incorporates side channels 20

5.1 Model of an enhanced smartcard with memory encryption mechanism (MEM) 50

6.1 Partitioning the address of requested data 75
6.2 Different types of cache memory 75
6.3 Basic structure of a time driven CBA 79
6.4 Basic structure of a trace driven CBA 81
6.5 Prime-and-Probe method 83
6.6 Formal outline of an access driven CBA 84
6.7 Combining small tables with permutation π 108

Bibliography

Acıiçmez, O., and Ç. K. Koç. 2006. Trace Driven Cache Attack on AES. In *ICICS*. P. Ning, S. Qing, and N. Li. (Eds.). Vol. 4307 of *Lecture Notes in Computer Science*. Springer Verlag. Pp. 112–121.

Acıiçmez, O., W. Schindler, and Ç. K. Koç. 2005. Improving Brumley and Boneh timing attack on unprotected SSL implementations. In *ACM Conference on Computer and Communications Security*. V. Atluri, C. Meadows, and A. Juels. (Eds.). ACM. Pp. 139–146.

Akkar, M.-L., and C. Giraud. 2001. An Implementation of DES and AES, Secure against Some Attacks. *In* Koç, Naccache and Paar (2001). Pp. 309–318.

Akkar, M.-L., and L. Goubin. 2003. A generic protection against high-order differential power analysis. *In* Johansson (2003). Pp. 192–205.

Akkar, M.-L., R. Bévan, and L. Goubin. 2004. Two Power Analysis Attacks against One-Mask Methods. In *11th International Workshop on Fast Software Encryption — FSE 2004*. B. K. Roy, and W. Meier. (Eds.). Vol. LNCS 3017 of *Lecture Notes in Computer Science*. Springer-Verlag.

Anderson, R. 2001. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley & Sons.

Anderson, R. J., and M. G. Kuhn. 1996. Tamper resistance – a cautionary note. *Proceedings of the second USENIX Workshop on Electronic Commerce*. USENIX Association. Pp. 1–11.

Bar-El, H., H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*. Vol. 94. Pp. 370–382.

Baudron, O., F. Boudot, P. Bourel, E. Bresson, J. Corbel, L. Frisch, H. Gilbert, M. Girault, L. Goubin, J.-F. Misarsky, P. Nguyen, J. Patarin, D. Pointcheval, J. Stern, J. Traor, and G. Poupard. 2000. GPS - An Asymmetric Identification Scheme for on the fly Authentication of Low Cost Smart Cards.

Bernstein, D. J. 2005. Cache-timing attacks on AES. `http://cr.yp.to/papers.html#cachetiming`.

Bertoni, G., V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. 2005. AES Power Attack Based on Induced Cache Miss and Countermeasure. *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*. IEEE Computer Society. Pp. 586–591.

Biham, E., and A. Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*. Burton S. Kaliski Jr. (ed.). Vol. 1294 of *Lecture Notes in Computer Science*. Springer. Pp. 513–525.

Biham, E., and A. Shamir. 1999. Power Analysis of the Key Scheduling of the AES Candidates. *Proceedings of the Second AES Candidate Conference (AES2)*. Rome, Italy.

Blömer, J., and J.-P. Seifert. 2003. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In *Financial Cryptography, 7th International Conference, FC 2003, Guadeloupe, French West Indies, January 27-30, 2003, Revised Papers*. R. N. Wright (ed.). Vol. 2742 of *Lecture Notes in Computer Science*. Springer-Verlag. Pp. 162–181.

Blömer, J., and V. Krummel. 2006. Fault based collision attacks on AES. In *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*. L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert. (Eds.). Vol. 4236 of *Lecture Notes in Computer Science*. Springer. Pp. 106–120.

Blömer, J., and V. Krummel. 2007. Analysis of countermeasures against access driven cache attacks on AES. In *Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers*. C. Adams, A. Miri, and M. Wiener. (Eds.). Lecture Notes in Computer Science. to appear.

Blömer, J., J. Guajardo, and V. Krummel. 2004. Provably secure masking of AES. In *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*. H. Handschuh, and M. A. Hasan. (Eds.). Vol. 3357 of *Lecture Notes in Computer Science*. Springer Verlag.

Boneh, D., R. A. DeMillo, and R. J. Lipton. 1997. On the importance of checking cryptographic protocols for faults (extended abstract). In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*. W. Fumy (ed.). Vol. 1233 of *Lecture Notes in Computer Science*. Springer. Pp. 37–51.

Brickell, E. F., G. Graunke, M. Neve, and J.-P. Seifert. 2006. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *Technical Report 2006/052*. Cryptology ePrint Archive. `http://eprint.iacr.org/2006/052`.

Brumley, D., and D. Boneh. 2005. Remote timing attacks are practical. *Computer Networks* **48**, 701–716.

Cathalo, J., F. Koeune, and J.-J. Quisquater. 2003. A New Type of Timing Attack: Application to GPS. *In* Walter, Koç and Paar (2003). Pp. 291–303.

Chari, S., C. S. Jutla, J. R. Rao, and P. Rohatgi. 1999. Towards sound approaches to counteract power-analysis attacks. *In* Wiener (1999). Pp. 398–412.

Chen, C.-N., and S.-M. Yen. 2003. Differential fault analysis on AES key schedule and some countermeasures. In *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia, July 9-11, 2003, Proceedings*. R. Safavi-Naini, and J. Seberry. (Eds.). Vol. 2727 of *Lecture Notes in Computer Science*. Springer. Pp. 118–129.

Clavier, C., J. Coron, and N. Dabbous. 2000. Differential Power Analysis in the Presence of Hardware Countermeasures. *In* Koç and Paar (2000). Pp. 252–263.

Daemen, J., and V. Rijmen. 1999. Resistance against Implementation Attacks: A comparative Study of the AES Proposals. *Second Advanced Encryption Standard (AES) Candidate Conference.* `http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm`.

Daemen, J., and V. Rijmen. 2002. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer Verlag.

Dhem, J.-F., F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. 1998. A practical implementation of the timing attack. In *CARDIS*. J.-J. Quisquater, and B. Schneier. (Eds.). number 1820 in *Lecture Notes in Computer Science*. Springer Verlag.

Diffie, W., and M. E. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* **22**, 644–654.

Drolet, G. 1998. A New Representation of Elements of Finite Fields $GF(2^m)$ Yielding Small Complexity Arithmetic Circuits. *IEEE Transactions on Computers* **47**, 938–946.

Dusart, P., G. Letourneux, and O. Vivolo. 2003. Differential fault analysis on A.E.S.. In *Applied Cryptography and Network Security, First International Conference, ACNS 2003. Kunming, China, October 16-19, 2003, Proceedings*. J. Zhou, M. Yung, and Y. Han. (Eds.). Vol. 2846 of *Lecture Notes in Computer Science*. Springer. Pp. 293–306.

van Eck, W. 1985. Electromagnetic radiation from video display units: an eavesdropping risk?. *Computers & Security* **4**, 269–286.

ElGamal, T. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara,* *California, USA, August 19-22, 1984, Proceedings*. G. R. Blakley, and D. Chaum. (Eds.). Vol. 196 of *Lecture Notes in Computer Science*. Springer-Verlag New York, Inc.. New York, NY, USA. Pp. 10–18.

Ferguson, N., and B. Schneier. 2003. *Practical Cryptography*. John Wiley & Sons.

Fournier, J., S. Moore, H. Li, R. Mullins, and G. Taylor. 2003. Security Evaluation of Asynchronous Circuits. *In* Walter et al. (2003). Pp. 125–136.

Gandolfi, K., C. Mourtel, and F. Olivier. 2001. Electromagnetic analysis: Concrete results. *In* Koç et al. (2001). Pp. 251–261.

von zur Gathen, J., and J. Gerhard. 2003. *Modern Computer Algebra*. 2nd ed. Cambridge University Press, Cambridge, UK,.

von zur Gathen, J., and M. Nöcker. 1997. Exponentiation in Finite Fields: Theory and Practice. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 12th International Symposium, AAECC-12, Toulouse, France, June 23-27, 1997, Proceedings*. T. Mora, and H. F. Mattson. (Eds.). Vol. 1255 of *Lecture Notes in Computer Science*. Springer. Pp. 88–113.

Giraud, C. 2004. DFA on AES. In *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers*. H. Dobbertin, V. Rijmen, and A. Sowa. (Eds.). Vol. 3373 of *Lecture Notes in Computer Science*. Springer. Pp. 27–41.

Golić, J. D. 2003. DeKaRT: A New Paradigm for Key-Dependent Reversible Circuits. *In* Walter et al. (2003). Pp. 98–112.

Golić, J. D., and C. Tymen. 2002. Multiplicative masking and power analysis of AES. *In* Kaliski Jr., Koç and Paar (2003). Pp. 198–212.

Goubin, L., and J. Patarin. 1999. DES and Differential Power Analysis, "The Duplication Method". In *Workshop on Cryptographic Hardware and Embedded Systems — CHES 1999*. Ç. K. Koç, and C. Paar. (Eds.). Vol. LNCS 1717 of *Lecture Notes in Computer Science*. Springer-Verlag. Pp. 158–172.

Guajardo, J., and C. Paar. 2002. Itoh-Tsujii Inversion in Standard Basis and Its Application in Cryptography and Codes. *Design, Codes, and Cryptography* **25**, 207–216.

Handy, J. 1998. *The Cache Memory Book: THE authorative reference on cache design*. 2nd ed. Academic Press.

Hennesey, J., and D. Patterson. 2002. *Computer Architecture: A Quantitative Approach*. 3rd ed. Morgan Kaufmann.

Hevia, A., and M. A. Kiwi. 1999. Strength of two data encryption standard implementations under timing attacks. *ACM Transactions on Information and System Security (TISSEC)* **2**, 416–437.

Hu, W.-M. 1992. Lattice scheduling and covert channels. *IEEE Symposium on Security and Privacy*. IEEE Press. Pp. 52–61.

Intel 1997. Using the RDTSC Instruction for Performance Monitoring. Intel Corporation 1997.

Intel 2006. *Intel© 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*.

ISO 2002. International Organization for Standardization, ISO/IEC 7816-3: Electronic signals and transmission protocols.

Itoh, T., and S. Tsujii. 1988. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Information and Computation* **78**, 171–177.

Johansson, T. (Ed.) 2003. *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers*. Vol. 2887 of *Lecture Notes in Computer Science*. Springer.

Kahn, D. 1996. *The Codebreakers*. Scribner.

Kaliski Jr., B. S., Ç. K. Koç, and C. Paar. (Eds.) 2003. *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Vol. 2523 of *Lecture Notes in Computer Science*. Springer.

Kelsey, J., B. Schneier, D. Wagner, and C. Hall. 1998. Side channel cryptanalysis of product ciphers. In *Computer Security - ESORICS 98, 5th European Symposium on Research in Computer Security, Louvain-la-Neuve, Belgium, September 16-18, 1998, Proceedings*. J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann. (Eds.). Vol. 1485 of *Lecture Notes in Computer Science*. Springer. Pp. 97–110.

Kerckhoffs, A. 1883. La cryptographie militaire. *Journal des sciences militaires* **IX**, 5–83 & 161–191.

Koç, Ç. K., and C. Paar. (Eds.) 2000. *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*. Vol. 1965 of *Lecture Notes in Computer Science*. Springer.

Koç, Ç., D. Naccache, and C. Paar. (Eds.) 2001. *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Vol. 2162 of *Lecture Notes in Computer Science*. Springer.

Kocher, P. C. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. N. Koblitz (ed.). Vol. 1109 of *Lecture Notes in Computer Science*. Springer. Pp. 104–113.

Kocher, P. C., J. Jaffe, and B. Jun. 1998. Introduction to Differential Power Analysis and Related Attacks. *Technical Report*. Cryptography Research, Inc.

Kocher, P. C., J. Jaffe, and B. Jun. 1999. Differential power analysis. *In* Wiener (1999). Pp. 388–397.

Koeune, F., and J.-J. Quisquater. 1999. A timing attack against Rijndael. *Technical Report CG-1999/1*. Université Catholique de Louvain.

Kömmerling, O., and M. G. Kuhn. 1999. Design principles for tamper-resistant smartcard processors. *Proceedings of the USENIX Workshop on Smartcard Technology — Smartcard '99*. USENIX Association. Pp. 9–20.

Kuhn, M. G. 2002. Optical Time-Domain Eavesdropping Risks of CRT Displays. *IEEE Symposium on Security and Privacy*. Pp. 3–18.

Kuhn, M. G. 2003. Compromising emanations: eavesdropping risks of computer displays. *Technical Report UCAM-CL-TR-577*. University of Cambridge.

Lenstra, H. W. 2002. Rijndael for algebraists.
`http://www.math.berkeley.edu/~hwl/papers/rijndael0.pdf`.

Lidl, R., and H. Niederreiter. 1983. *Finite Fields*. number 20 in *Encyclopedia of Mathematics and its Applications*. Addison Wesley.

Mangard, S. 2002. A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. *Proceedings of the 5th International Conference on Information Security and Cryptology (ICISC 2002)*. Vol. LNCS 2587. Springer-Verlag. Pp. 343–358.

May, M., H. Muller, and N. Smart. 2001a. Non-Deterministic Processors. *6th Australian Conference On Information Security and Privacy (ACISP)*. Pp. 115–129.

May, M., H. Muller, and N. Smart. 2001b. Random Register Renaming to Foil DPA. *In* Koç et al. (2001).

Menezes, A. J., P. C. van Oorschot, and S. A. Vanstone. 1997. *Handbook of Applied Cryptography*. CRC Press.

Messerges, T. S. 2000. Securing the AES finalists against power analysis attacks. In *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA,* *April 10-12, 2000, Proceedings*. B. Schneier (ed.). Vol. 1978 of *Lecture Notes in Computer Science*. Springer. Pp. 150–164.

Montgomery, P. L. 1985. Modular multiplication without trial division. *Mathematics of Computation* **44**, 519–521.

Moore, S., R. Anderson, R. Mullins, G. Taylor, and J. Fournier. 2003. Balanced Self-Checking Asynchronous Logic for Smart Card Applications. *Journal of Microprocessors and Microsystems* **27**, 421–430.

Neve, M., and J.-P. Seifert. 2006. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Quebec, Canada, August 17 & 18, 2006, Revised Selected Papers*. E. Biham, and A. Youssef. (Eds.). to appear.

NIST 2001. Announcing the ADVANCED ENCRYPTION STANDARD (AES). *FIPS-PUB 197*. National Institute for Standards and Technology (NIST).

OpenSSL Project 2005. http://www.openssl.org.

Örs, S., F. Gürkaynak, E. Oswald, and B. Preneel. 2004. Power-Analysis Attack on an ASIC AES Implementation. *Proceedings of the 2004 International Symposium on Information Technology (ITCC 2004)*. IEEE Computer Society.

Osvik, D. A., A. Shamir, and E. Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. D. Pointcheval (ed.). Vol. 3860 of *Lecture Notes in Computer Science*. Springer. Pp. 1–20.

Otto, M. 2005. *Fault Attacks and Countermeasures*. PhD thesis. University of Paderborn.

Page, D. 2002. Theoretical use of cache memory as a cryptanalytic side-channel. *Technical Report CSTR-02-003*. Department of Computer Science, University of Bristol.

Page, D. 2003. Defending against cache based side-channel attacks. *Information Security Technical Report* **8**, 30–44.

Percival, C. 2005. Cache missing for fun and profit. `www.daemonology.net/papers/htt.pdf`.

Piret, G., and J.-J. Quisquater. 2003. A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. *In* Walter et al. (2003). Pp. 77–88.

Quisquater, J.-J., and D. Samyde. 2001. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*. I. Attali, and T. P. Jensen. (Eds.). Vol. 2140 of *Lecture Notes in Computer Science*. Springer. Pp. 200–210.

Quisquater, J.-J., and D. Samyde. 2002. Eddy current for magnetic analysis with active sensor. *E-Smart 2002, Nice, France*.

Rankl, W., and W. Effing. 2002. *Handbuch der Chipkarten*. 4. ed. Carl Hanser Verlag.

Rivest, R. L., A. Shamir, and L. M. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems.. *Communications of the ACM (CACM),* **21**, 120–126.

Satoh, A., S. Morioka, K. Takano, and S. Munetoh. 2001. A Compact Rijndael Hardware Architecture with S-Box Optimization. In *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*. C. Boyd (ed.). Vol. LNCS 2248 of *Lecture Notes in Computer Science*. Springer-Verlag. Pp. 239–254.

Schindler, W. 2000. A timing attack against RSA with the Chinese Remainder Theorem. *In* Koç and Paar (2000). Pp. 109–124.

Schneier, B. 1996. *Applied Cryptography*. John Wiley & Sons.

Schramm, K., G. Leander, P. Felke, and C. Paar. 2004. A collision-attack on AES: Combining side channel- and differential-attack. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. M. Joye, and J.-J. Quisquater. (Eds.). Vol. 3156 of *Lecture Notes in Computer Science*. Springer. Pp. 163–175.

Schramm, K., T. J. Wollinger, and C. Paar. 2003. A new class of collision attacks and its application to DES. *In* Johansson (2003). Pp. 206–222.

Shamir, A., and E. Tromer. 2004. Acoustic cryptanalysis - on noisy people and noisy machines. http://theory.csail.mit.edu/~tromer/acoustic/.

Shoup, V. 2005. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press.

Skorobogatov, S. P., and R. J. Anderson. 2002. Optical fault induction attacks. *In* Kaliski Jr. et al. (2003). Pp. 2–12.

Stallings, W. 2005. *Operating Systems: Internals and Design Principles*. 5th ed. Prentice Hall.

Stephenson, N. 1999. *Cryptonomicon*. 1st ed. Eos (HarperCollins).

Tiri, K., and I. Verbauwhede. 2003. Securing Encryption Algorithms against DPA at the Logic Level: Next Generation Smart Card Technology. *In* Walter et al. (2003). Pp. 125–136.

Tiri, K., M. Akmal, and I. Verbauwhede. 2002. A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards. *28th European Solid-State Circuits Conference (ESSCIRC 2002).*

Tiu, C. C. 2005. *A new frequency-based side channel attack for embedded systems.* Master's thesis. University of Waterloo.

Trichina, E. 2003. Combinational Logic Design For AES SubByte Transformation on Masked Data. *Cryptology eprint archive: Report 2003/236.* IACR.

Trichina, E., D. D. Seta, and L. Germani. 2002. Simplified Adaptive Multiplicative Masking for AES. *In* Kaliski Jr. et al. (2003). Pp. 187–197.

Trostle, J. T. 1998. Timing attacks against trusted path. *IEEE Symposium on Security and Privacy.* IEEE Press. Pp. 125–134.

Tsunoo, Y., E. Tsujihara, K. Minematsu, and H. Miyauchi. 2002. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. *International Symposium on Information Theory and Its Applications (ISITA).*

Tsunoo, Y., E. Tsujihara, M. Shigeri, H. Kubo, and K. Minematsu. 2006. Improving cache attacks by considering cipher structure. *International Journal of Information Security (IJIS)* **5**, 166–176.

Tsunoo, Y., H. Kubo, M. Shigeri, E. Tsujihara, and H. Miyauchi. 2003a. Timing attack on AES using cache delay in S-boxes. *Symposium on Cryptography and Information Security.*

Tsunoo, Y., T. Kawabata, E. Tsujihara, K. Minematsu, and H. Miyauchi. 2003b. Timing attack on KASUMI using cache delay in S-boxes. *Symposium on Cryptography and Information Security.*

Tsunoo, Y., T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. 2003c. Cryptanalysis of DES Implemented on Computers with Cache. *In* Walter et al. (2003). Pp. 62–76.

Tsunoo, Y., T. Suzaki, T. Saito, T. Kawabata, and H. Miyauchi. 2003d. Timing attack on Camellia using cache delay in S-boxes. *Symposium on Cryptography and Information Security.*

Voigtländer, P. 2003. Entwicklung einer Hardwarearchitektur für einen AES-Coprozessor. *Diplomarbeit.* Fachbereich Informatik, Mathematik und Naturwissenshaften, Technische Informatik, HTWK Leipzig. Germany.

Walter, C. D., Ç. K. Koç, and C. Paar. (Eds.) 2003. *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*. Vol. 2779 of *Lecture Notes in Computer Science*. Springer.

Wiener, M. J. (Ed.) 1999. *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Vol. 1666 of *Lecture Notes in Computer Science*. Springer.

Wright, P. 1987. *Spy Catcher: The Candid Autobiography of a Senior Intelligence Officer*. Viking Adult.

Yang, B., K. Wu, and R. Karri. 2004. Scan based side channel attack on dedicated hardware implementations of data encryption standard. *ITC '04: Proceedings of the International Test Conference on International Test Conference*. IEEE Computer Society. Washington, DC, USA. Pp. 339–344.