ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta jaderná a fyzikálně inženýrská

DIPLOMOVÁ PRÁCE

Praha, 2016 Jakub Klemsa

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta jaderná a fyzikálně inženýrská

Katedra matematiky

DIPLOMOVÁ PRÁCE

Analýza AES white-box schémat pomocí útoku postranním kanálem

Side-Channel Attack Analysis of AES White-Box Schemes

Vypracoval:	Jakub Klemsa
Školitel:	prof. RNDr. Václav Matyáš, M.Sc., Ph.D.
Akademický rok:	2015/2016

Na toto místo přijde svázat **zadání mé diplomové práce**!

V jednom z výtisků musí být **originál** zadání, v ostatních kopie.

Čestné prohlášení

Prohlašuji na tomto místě, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze dne 6. května 2016

. . . . . .
Jakub Klemsa

Poděkování

Děkuji prof. RNDr. Václavu Matyáši, M.Sc., Ph.D., za vedení mé diplomové práce a za podnětné návrhy, které ji obohatily.

Jakub Klemsa

Název práce: **Analýza AES white-box schémat pomocí útoku postranním kanálem**

Autor: Jakub Klemsa

Obor: Matematická informatika

Druh práce: Diplomová práce

Vedoucí práce: prof. RNDr. Václav Matyáš, M.Sc., Ph.D., Fakulta informatiky, Masarykova univerzita

Konzultant: Mgr. Dušan Klinec

Abstrakt: Nejenže jsou všechna současná vědecká white-box schémata standardizovaných šifer matematicky zlomená, další ránu pro ně představuje nový útok od autorů Bos a kol. představený v červenci 2015 ve článku *Differential Computation Analysis: Hiding your White-Box Designs is Not Enough.* Tento útok je velmi univerzální a účinný – nepotřebuje ani znalost implementace, ani použití reverzního inženýrství.
V této práci se mi podařilo reprodukovat útok na implementaci konkrétní chráněné šifry, vylepšit ho a nakonec i vysvětlit jeho podstatu, což původní autoři nezmiňují.

Klíčová slova: Stopa v paměti, útok postranním kanálem, white-box AES.

Title: **Side-Channel Attack Analysis of AES White-Box Schemes**

Author: Jakub Klemsa

Abstract: Not only have all current scientific white-box schemes of standardized ciphers been mathematically broken, they further face a novel attack introduced by Bos et al. in July, 2015, in *Differential Computation Analysis: Hiding your White-Box Designs is Not Enough.* This attack is very universal and powerful – it requires neither knowledge of the implementation, nor use of reverse engineering techniques.
In this thesis, we successfuly reproduced this attack against an implementation of a specific protected cipher. We further improved the original attack, and finally provided an explanation of its principle, which the original authors omitted.

Keywords: Memory trace, side-channel attack, white-box AES.

Table of Contents

Notations and Symbols

$\|S\|$	Cardinality of the set S
$\Pr(\omega)$	Probability of the event ω
$k \xleftarrow{R} K$	k is taken uniformly random from the set K
$a \oplus b$	Bitwise XOR of two bitstrings a and b
$a \parallel b$	Concatenation of two strings a and b
$\mathbb{N}$	Positive integers
$\mathbb{R}$	Real numbers
P	Deterministic polynomial complexity class, i.e., the set of decision problems which can be solved by a deterministic Turing machine in polynomial time
NP	Non-deterministic polynomial complexity class, i.e., the set of decision problems which can be solved by a non-deterministic Turing machine in polynomial time
$\mathrm{GF}(p^n)$	Galois Field with p^n elements, p prime
$F[x]$	Ring of polynomials over the ring F

Preface

Cryptography emerged long long time ago with the need for confidential communication in the presence of enemies – Caesar cipher can serve as a well-known example of an ancient cipher. Cryptography also had considerable consequences in the past – for instance, Mary, Queen of Scots, was executed back in 16^{th} century, because she was planning to assassinate Queen Elizabeth I of England and her cipher got broken. However, the most appreciable success of cryptography was definitely the breach of Nazi Germany's Enigma code, which they fortunately believed to be unbreakable.

Since its revival few decades ago, cryptography became a proper discipline of modern mathematics and now it covers much more than "hiding secrets". Indeed, there are cryptographic primitives that allow us to construct digital signatures, message authentication codes or even more complicated structures like digital currencies.

In this thesis, we will focus on another interesting problem: let us imagine that we would like to work with confidential data in an untrusted environment. For example, we would like to let an untrusted party process our confidential data without learning anything about it, or – which we will study in this thesis – we would like to let anybody use our cryptographic key without being able to recover it. Such a property of a cipher implementation will be referred to as *white-box attack resistance*. However, white-box attack resistance appears to be quite hard to achieve.

The primary goal of this thesis is to reproduce, improve and analyze a recently introduced attack technique [12] against white-box attack protected implementations.

Work Overview

Chapter 1: Introduction to White-Box Cryptography.
First of all, we give an introduction to cryptography and white-box cryptography. Then we present the structure of Advanced Encryption Standard (AES) – a cipher that we will be particularly interested in during the whole thesis. Finally, we conclude this chapter with a description of a particular design of a white-box variant of AES.

Chapter 2: Using Side-Channel Attack Tools.
Next we present an attack technique that originally targets hardware leakage. However, it can be advantageously used in software scenario as well – we describe a recent attack based on this idea.

Chapter 3: Results & Improvements.
In the third chapter, we first reproduce previous results of the attack, then we suggest our improvements and run the attack again. Finally, we introduce an approach that unifies our contributions.

Chapter 4: Analysis of the Attack against WBAES.
In the fourth chapter, we present some remarks regarding the attack, then we exploit them to suggest a blind attack (i.e., an attack without knowing the actual key). We also mention some consequences in white-box implementation design. To conclude this chapter, we outline, justify and confirm our explanation of the attack.

Chapter 5: Future Work.
Finally, we summarize possible topics of further research.

Conclusion.

Chapter 1

Introduction to White-Box Cryptography

Before we introduce the concept of white-box cryptography, we present some introductory topics of cryptography itself – we especially formalize notions of security. Then we describe the Advanced Encryption Standard [47] and an attempt for its white-box variant [16].

1.1 Introduction to Cryptography

In cryptography, the most typical situation is where Alice wants to communicate with Bob confidentially over an insecure channel, while Eve may eavesdrop on the channel. In the beginning, let us suppose that Alice and Bob share a certain portion of secret information referred to as *secret key*, and both control a secure execution environment, where they intend to run their ciphering algorithm, so that Eve can only observe what they send over the channel.

Note 1.1. A list of such attacker's abilities is referred to as *threat model*. Note that in the previous threat model, the attacker is quite weak – she can only read the traffic. If she were for instance able to amend the traffic, Alice and Bob would need to introduce other features to their protocol, so that they could detect forged messages.

Before we begin with introducing any means of security, we always need to identify the threat model first. For now, we will only consider eavesdropping.

Our first intuitive requirements on Alice's and Bob's encryption algorithm could be expressed as follows: no matter what plaintext Alice and Bob encrypt and send over the channel, Eve shall *not* be able to

1. recover any portion of any plaintext,
2. recover the secret key.

The item 1 requirement is also referred to as *confidentiality*. Note that the item 2 requirement – key recovery – would lead to plaintext recovery as well, on the other

hand, not necessarily the other way around. Let us give an example of a primitive cipher.

Example 1.2. *Shift cipher* [38] is an ancient cipher already used by Julius Caesar and it works as follows: it inputs a string of letters from a finite alphabet $\mathcal{A}$, maps its letters bijectively to the numbers from 0 to $|\mathcal{A}| - 1$ and adds a constant $c \in \{0, \ldots, |\mathcal{A}| - 1\}$ modulo $|\mathcal{A}|$ to each number. Finally, it maps the numbers back to letters, yielding a string "shifted" by c, where c can be interpreted as an element of $\mathcal{A}$ as well. Decryption can be achieved simply by subtracting c from the ciphertext in a similar way.

Shift cipher can be easily broken by pen and paper only, since there are as many keys as letters in the alphabet, typically 26 English letters. We simply try them all until the plaintext makes sense. Such an approach is referred to as *brute force attack* or *exhaustive search*.

As we could have seen in the previous example, shift cipher is not very secure cipher. Therefore we would like to formalize the concept of a cipher and its security.

1.1.1 Symmetric And Asymmetric Cipher

There are two large families of ciphers – symmetric and asymmetric ones. Shift cipher, introduced in Example 1.2, belongs to the symmetric family, which follows from the fact that it uses the same key for both encryption and decryption, while asymmetric ciphers use distinct keys for encryption and decryption. Asymmetric ciphers are also referred to as public key ciphers, but we will only consider symmetric ciphers in this thesis; a definition follows.

Definition 1.3 (Symmetric Cipher)**.** Let K denote the key space, M the message space and C the ciphertext space. *Symmetric cipher* is a pair of efficiently computable (possibly randomized) functions (E, D), $E : K \times M \to C$, $D : K \times C \to M$, such that $\forall m \in M$ and $\forall k \in K$ it holds

$$\Pr\Big(D\big(k, E(k, m)\big) = m\Big) = 1. \tag{1.1}$$

Note 1.4. Going back to Example 1.2, the key space is $\{0, \ldots, |\mathcal{A}| - 1\} \sim \mathcal{A}$, the message and ciphertext spaces are $\mathcal{A}^*$ which stands for the set of all strings over the alphabet $\mathcal{A}$.

According to the previous definition, we do not seem to have a definition of a proper cipher at all – the only requirement is correctness (cf. Equation 1.1), but it totally lacks any security requirement. As we will see later, it is actually quite complicated to define "security" properly.

1.1.2 Information-Theoretic Approach

Shannon's groundbreaking work [52] on the mathematics of communication gives us a tool – information-theoretic approach. We can then define a secure cipher as a cipher, for which it holds that its ciphertexts carry no information about respective plaintexts. Note that this can be rewritten in a more friendly form, a definition follows.

Definition 1.5 (Perfectly Secure Cipher)**.** Let (E, D) be a symmetric cipher. Then it is called *perfectly secure* if $\forall m_0, m_1 \in M$ such that $|m_0| = |m_1|$ and $\forall c \in C$ it holds $\Pr(E(k, m_0) = c) = \Pr(E(k, m_1) = c)$, where k is uniformly random in K, denoted by $k \xleftarrow{R} K$.

Note 1.6. In other words, the attacker is not able to *distinguish* encryption of m_0 from encryption of m_1 by *any* means.

There is a cipher that is perfectly secure, see the following example.

Example 1.7 (Vernam Cipher)**.** Let $K = M = C = \{0,1\}^n$ and $k \xleftarrow{R} K$. Given a plaintext m, *Vernam cipher* applies bitwise XOR of the key k and the plaintext m, yielding a ciphertext c, i.e., $c = m \oplus k$.

Note 1.8. Vernam cipher is also referred to as *One Time Pad* (OTP). Note that it is crucial that each key is only used once, otherwise we could attack the cipher easily. Indeed, given two ciphertexts using the same key, i.e., $c_0 = m_0 \oplus k$ and $c_1 = m_1 \oplus k$, one can compute $c_0 \oplus c_1 = m_0 \oplus m_1$. This can be practically attacked using some a priori knowledge about the plaintexts (e.g., encoding – ASCII, format and language of the plaintext – XML and English, and so on), which reduces the amount of possibilities and typically leads to a complete plaintext recovery.

Perfect security seems to be exactly what we want from a secure cipher, but the following theorem shows that it implies certain inconveniences.

Theorem 1.9. Let (E, D) be a perfectly secure cipher. Then $|K| \geq |M|$.

Note 1.10. This consequence of perfect security apparently goes against our intention – we would like to establish a shared secret key of a limited length and then, using the key, be able to encrypt as much information as we want. For this reason, we need to weaken our security assumption. Note that previously, there was no assumption on attacker's computing power – the requirement was actually to hide the plaintext from *any* attacker, i.e., including an attacker with unlimited computing power.

1.1.3 Complexity-Theoretic Approach

We would like to utilize the fact that the attacker typically does not possess unlimited computing power, but she can only solve probabilistic polynomial problems.

Note 1.11. There arises a natural question: *Probabilistic polynomial in terms of what?* In cryptography, there is a meter for that referred to as *security parameter*, which is typically the key-length. In the following text and for this purpose, the expression *effective* will be used frequently.

With this assumption, we can weaken our requirement on ciphertext from "carries no information" to "is computationally impossible to gain some information". In a manner analogous to Note 1.6, we would like to make impossible to distinguish which plaintext has been encrypted, now considering an effective attacker only. Let us present this idea in the following game.

Game 1.12. There are two parties – a *challenger* and an *adversary*. First, the challenger chooses a random key $k \xleftarrow{R} K$ and a random bit $b \xleftarrow{R} \{0,1\}$. Then the adversary sends two *equally long* plaintexts m_0, m_1 of her choice to the challenger. The challenger encrypts m_b (according to b she has chosen) and sends the resulting ciphertext c back to the adversary. The adversary is allowed to perform several such queries while the challenger keeps constant key k and bit b; the adversary is only limited by polynomial time.

The aim of the adversary is to distinguish effectively which b has been chosen by the challenger; see Figure 1.1 depicting this game.

Note 1.13. Resistance of a cipher to this kind of attack is referred to as *semantic security*.

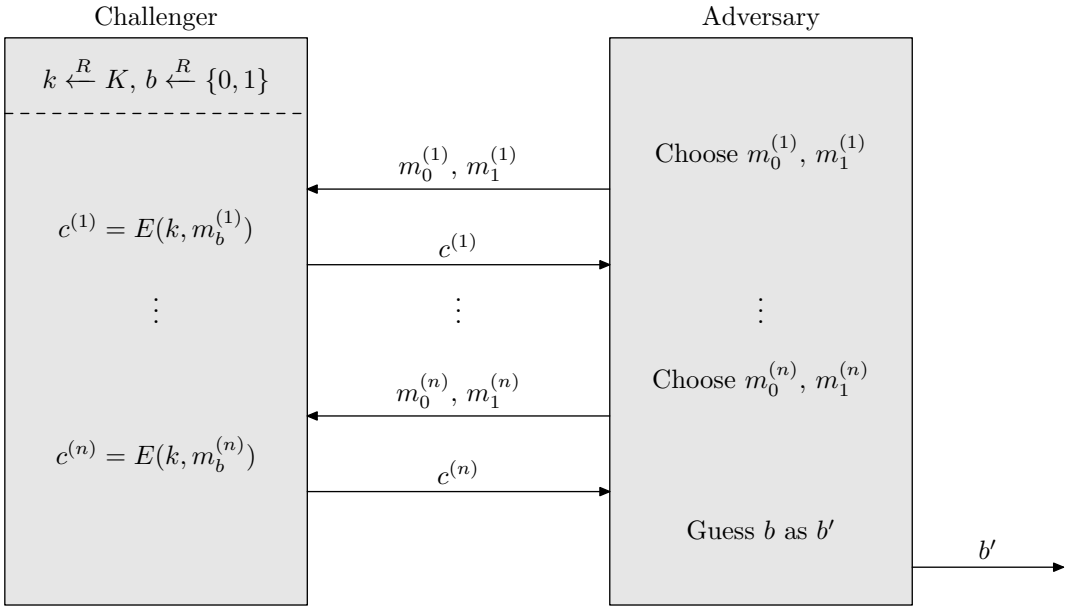

Figure 1.1: Game 1.12 depicted.

Note 1.14. The adversary is intentionally allowed to provide the same plaintext over and over again. However, it follows that the challenger has to encrypt the same plaintext differently each time, otherwise there would exist a trivially winning adversary.

Indeed, such an adversary would pick three distinct plaintexts m_0, m_1, m_2 and send (m_0, m_1) to the challenger, while she would get back some ciphertext c_a. In her second query, the adversary would send (m_0, m_2) to the challenger and receive another ciphertext, say c_b. If encryptions of m_0 were equal, she could decide with certainty:

- $b' = 0$ if $c_a = c_b$, or
- $b' = 1$ otherwise,

and win the game.

Note 1.15. In order to achieve semantic security of a cipher, the cipher *must* encrypt the same plaintext differently each time. This can be achieved using randomization or counters, both approaches are practically used.

Definition of Semantic Security

Let us introduce some notion first.

Definition 1.16 (Negligible Function)**.** Let $\varepsilon : \mathbb{N} \to \mathbb{R}$. ε is called *negligible* if $\forall d \in \mathbb{N}$ $\exists \lambda_0 \in \mathbb{N}$ such that $\forall \lambda > \lambda_0$ it holds $\varepsilon(\lambda) \leq \frac{1}{\lambda^d}$. In the opposite case, ε is called *non-negligible*.

Definition 1.17 (Overwhelming Function)**.** Let $\varepsilon : \mathbb{N} \to \mathbb{R}$. ε is called *overwhelming* if $1 - \varepsilon$ is negligible.

Note 1.18. In practice, *negligible* is commonly used with concrete constants as well: $\varepsilon < \frac{1}{2^{80}}$ is considered negligible, i.e., events with such probability are not considered to occur during our lives. On the other hand, $\varepsilon > \frac{1}{2^{30}}$ is considered non-negligible, i.e., it is likely to observe an event with such probability.

Definition 1.19 (Oracle)**.** Let $F : X \to Y$. *Oracle evaluating* F is a device that, given $x \in X$, evaluates and returns $F(x)$ in unit time; denoted by $\mathcal{O}_F$.

Definition 1.20 (Oracle with Limited Access)**.** Let $F : X \to Y$ and $n \in \mathbb{N}$. *Oracle with limited access evaluating* F is an oracle evaluating F, where the number of queries is limited to n; denoted by $\mathcal{O}_F^{(n)}$.

Definition 1.21 (Statistical Test)**.** Let $F : X \to Y$. *Statistical test* is a (possibly randomized) effective algorithm A, which only has an access to an oracle evaluating F, and which outputs 0 or 1; denoted by $A(\mathcal{O}_F)$.

Note 1.22. Statistical test will be also referred to as *adversary*.

Definition 1.23 (Advantage)**.** Let $\mathcal{F}, \mathcal{G}$ be two distinct subsets of the set of all functions from X to Y, and let A be an adversary. We define *advantage* of adversary A as

$$\mathrm{Adv}(A, \mathcal{F}, \mathcal{G}) = \left| \Pr_{F \xleftarrow{R} \mathcal{F}} \Big(A(\mathcal{O}_F) = 1 \Big) - \Pr_{G \xleftarrow{R} \mathcal{G}} \Big(A(\mathcal{O}_G) = 1 \Big) \right|.$$

Definition 1.24 (Advantage with Limited Access)**.** Let $n \in \mathbb{N}$. *Advantage with limited access* is defined as advantage, where oracles with access limited to n are used instead; denoted by $\mathrm{Adv}^{(n)}$.

In other words, advantage tells us how likely adversary A is able to distinguish a random function from one set of functions from a random function from another set, based on its input/output behavior only. According to these sets, advantage can be "customized" for a specific purpose. Let us define such advantage for an adversary tempting to distinguish which of her plaintexts has been encrypted, i.e., tells us how likely she is to win the Game 1.12.

Definition 1.25 (Semantic Security Advantage)**.** Let $E : K \times M \to C$ be an encryption function of a symmetric cipher, $\mathcal{E}_b = \{E_b : M^2 \to C \mid E_b(m_0, m_1) = E(k, m_b), k \in K\}$ for $b = 0, 1$, and A an adversary. We define *semantic security advantage* of adversary A as

$$\mathrm{Adv}_{\mathrm{SS}}(A, E) = \mathrm{Adv}(A, \mathcal{E}_0, \mathcal{E}_1).$$

Semantic security advantage gives us a reasonable meter for formalization of Game 1.12. Let us use it to define another notion of security, cf. Definition 1.5 (Perfectly Secure Cipher).

Definition 1.26 (Semantically Secure Cipher)**.** Let (E, D) be a symmetric cipher. Then it is called *semantically secure* if, for *any* adversary A, the semantic security advantage $\mathrm{Adv}_{\mathrm{SS}}(A, E)$ is negligible.

Semantically secure cipher therefore cannot be broken by any probabilistic-polynomial adversary with overwhelming probability. On the other hand, it might be broken by an adversary with unlimited computing power – unlike a perfectly secure cipher that cannot be broken by any means; see Note 1.6.

The benefit of semantic security over perfect security is that semantic security does not imply any inconvenient consequence analogous to Theorem 1.9, i.e., addresses our intention as stated in Note 1.10, and practically provides similar security.

Let us now give a nice real-world example, how things can go wrong if semantic security is not complied, even though the design follows common sense.

Example 1.27. In 2012, a combination of settings in TLS[1] allowed researchers to hijack a web session by cracking encrypted cookies [23]. The problem arose when compression was followed by encryption, which actually makes sense. Let us have a look from theoretical point of view.

Remind the condition of Game 1.12 – it requires two plaintexts of equal length. With this assumption, the respective ciphertexts were indistinguishable. However, if the plaintext is compressed first, it can result in differently-sized inputs to the subsequent (semantically secure) cipher! This obviously harms the assumption of inputs of equal length, and may allow the adversary to distinguish the ciphertexts. Note that only very little queries (around six) were needed to attack one byte of an encrypted cookie in the practical attack!

This vulnerability was given a CVE[2] code CVE-2012-4929 and a human-readable name “CRIME” (Compression Ratio Info-leak Made Easy).

Shannon’s Principles

Before we present how semantic security is practically achieved, let us introduce a principle that is widely followed in cipher design.

[1] Transport Layer Security, a cryptographic protocol [21].

[2] Common Vulnerabilities and Exposures, a standard for information security vulnerability names, see `cve.mitre.org`.

Principle 1.28 (Shannon)**.** As outlined in another Shannon's work [51], secure ciphers are suggested to employ two basic principles – *confusion* and *diffusion*.

Confusion. The relation between information in plaintext or key and information in respective ciphertext shall be very complex, e.g., should not belong to a specific subset of mappings, especially to linear ones.

Diffusion. Small piece of information in plaintext or key shall be spread in a long piece of respective ciphertext, e.g., one bit of plaintext or key should change a large portion of respective ciphertext.

Semantic security is usually achieved using smaller building blocks, which are required to have certain properties – they especially make use of confusion and diffusion. The following section describes a building block of one of possible approaches.

1.1.4 Block Cipher

There are two main approaches how to construct a candidate semantically secure cipher. They make use of

1. a *stream cipher*, or
2. a *block cipher*,

respectively. Stream ciphers achieve semantic security by proper initialization, on the other hand, block ciphers use initialization together with *modes of operation*, e.g., *Cipher Block Chaining* (CBC) or *Counter Mode* (CTR). Modes of operation will not be covered in this thesis; only block ciphers and their desired properties will be described, a bunch of definitions follows.

Definition 1.29 (Pseudorandom Permutation)**.** Let $E : K \times X \to Y$. We call E a *pseudorandom permutation* (PRP) if there exists an efficient algorithm that evaluates E, the function $E(k, \cdot) : X \to Y$ is a bijection and there exists an efficient algorithm to compute $E^{-1}(k, \cdot)$, for every $k \in K$.

Note 1.30. Pseudorandom permutation is usually referred to as *block cipher*. Note that the previous definition is in some sense very similar to Definition 1.3 – it does not introduce any security requirements either.

Pseudorandom permutations will play the main role from now and k will play the role of their key. In order to achieve semantic security of certain constructions using a PRP, we need the PRP to behave indistinguishable from a truly random bijection. Let us demonstrate a possible approach on a game similar to Game 1.12.

Game 1.31. There are two parties – a *challenger* and an *adversary*. Let $E : K \times X \to Y$ be a PRP. First, the challenger chooses a random key $k \xleftarrow{R} K$, a random bijection $F : X \to Y$ and a random bit $b \xleftarrow{R} \{0, 1\}$. Then the adversary sends $x \in X$ of her choice to the challenger. The challenger applies $E(k, \cdot)$ or F if $b = 0$ or 1, respectively, and

sends the result back to the adversary. The adversary is allowed to perform several such queries.

The aim of the adversary is to distinguish effectively which b has been chosen by the challenger.

Before we give a formal definition of a secure PRP, we define adversary's advantage in the previous game in a manner analogous to Definition 1.25.

Definition 1.32 (PRP Advantage)**.** Let $E : K \times X \to Y$ be a PRP, $\mathcal{B}$ the set of all bijections from X to Y, and A an adversary. We define *PRP advantage* of adversary A as

$$\mathrm{Adv}_{\mathrm{PRP}}(A, E) = \mathrm{Adv}\Big(A, \{E(k, \cdot) \mid k \in K\}, \mathcal{B}\Big).$$

PRP advantage tells us how likely adversary A is able to distinguish a PRP with a random key from a truly random bijection based on its input/output behavior only. The definition of secure PRP follows.

Definition 1.33 (Secure PRP)**.** Let $E : K \times X \to Y$ be a PRP. Then it is called *secure PRP* if, for *any* adversary A, the PRP advantage $\mathrm{Adv}_{\mathrm{PRP}}(A, E)$ is negligible.

In other words, given a random key, PRP is secure if it is computationally indistinguishable from a truly random bijection.

An example covering most introduced concepts follows.

Example 1.34. Let $E : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$ be a PRP defined as

$$E(k, x) = \mathrm{bin}\big((k)_2 + (x)_2 \mod 2^{128}\big)$$

where $\mathrm{bin}(\cdot)$ stands for number's 128-bit binary representation and $(\cdot)_2$ stands for a number represented by given binary string.

Now let us construct an adversary A as follows: it generates two arbitrary but distinct $x_0, x_1 \in \{0,1\}^{128}$, and feeds them to an oracle computing either PRP E, or a truly random bijection. The oracle returns y_0 and y_1, respectively, and the adversary compares values $(y_i)_2 - (x_i)_2 \mod 2^{128}$ for $i = 0, 1$ with each other. If these values are equal, A returns 1, otherwise it returns 0.

Note that if the oracle computes PRP E, then these values are always equal. Otherwise – i.e., the oracle computes a truly random bijection – the values are distinct with overwhelming probability. Therefore the PRP advantage of such adversary is

$$\mathrm{Adv}_{\mathrm{PRP}}(A, E) = \left| \Pr_{k \xleftarrow{R} \{0,1\}^{128}} \Big(A(\mathcal{O}_{E(k,\cdot)}) = 1\Big) - \Pr_{F \xleftarrow{R} \mathcal{B}} \Big(A(\mathcal{O}_F) = 1\Big) \right| = 1 - \varepsilon,$$

where $\mathcal{B}$ stands for the set of all bijections on $\{0,1\}^{128}$ and ε is negligible.

It follows that such PRP is totally insecure, since there exists an adversary with overwhelming advantage, but only a negligible advantage is allowed.

Note 1.35. In the previous example, the adversary exploits the knowledge of the block cipher's internal structure – she knows everything but the key. One could have an idea to hide the structure from the adversary, but this is actually a *very* bad idea. Once our construction leaks, we need to deploy a new ciphering algorithm confidentially, but we have no way to do so. Moreover, note that a secure PRP was required to be resistant to *all* adversaries, hence hiding your design has absolutely no theoretical support.

We shall rather use a secure publicly known block cipher and only rely on the key. Once our key is compromised, we simply establish a new key. Also note that our key is only to be protected within our device, but the cryptographic algorithm would have to be protected globally.

Such an approach – hiding the design of a cipher – is referred to as *security by obscurity* and shall be strictly avoided.

This idea has already been addressed in 1883 by Kerckhoffs [5]. It is usually referred to as *Kerckhoffs' Principle* and its six rules follow (originally French, English translation taken from [45]):

Principle 1.36 (Kerckhoffs)**.**

1. The system must be substantially, if not mathematically, undecipherable;
2. The system must not require secrecy and can be stolen by the enemy without causing trouble;
3. It must be easy to communicate and retain the key without the aid of written notes, it must also be easy to change or modify the key at the discretion of the correspondents;
4. The system ought to be compatible with telegraph communication;
5. The system must be portable, and its use must not require more than one person;
6. Finally, given the circumstances in which such system is applied, it must be easy to use and must neither stress the mind or require the knowledge of a long series of rules.

Hence harming rule 2 from the previous list leads to security by obscurity as introduced in Note 1.35.

1.1.5 Output Indistinguishability of a Secure PRP

Now we know what a secure PRP is (or a secure block cipher) – it is based on Game 1.31, where the aim of the adversary is to distinguish a PRP from a truly random bijection. The obvious question is, what happens if we put a secure PRP into Game 1.12, where the adversary tempts to distinguish which of her two inputs has been processed.

Note that PRP *is* a cipher (cf. Definitions 1.3 and 1.29), so it makes sense. Indeed, given a PRP, there exists an effective algorithm evaluating it and its inverse, and also Equation 1.1 holds.

Note 1.37. PRP has no ambiguity, hence we will need to either limit the amount of adversary-challenger queries to one, or, equivalently, change the key after each query. If we did not do so, then there would obviously exist a trivially winning adversary as outlined in Note 1.14.

Let us modify Game 1.12 in a manner described in Note 1.37 for the purpose of testing PRPs.

Game 1.38. There are two parties – a *challenger* and an *adversary*. Let $E : K \times X \to Y$ be a PRP. First, the challenger chooses a random key $k \xleftarrow{R} K$ and a random bit $b \xleftarrow{R} \{0, 1\}$. Then the adversary sends two equally long plaintexts m_0, m_1 of her choice to the challenger. The challenger computes $c = E(k, m_b)$ and sends it back to the adversary. The adversary is limited to one query only.

The aim of the adversary is to distinguish effectively which b has been chosen by the challenger.

Based on the previous game, let us define *PRP semantic security advantage* and *semantically secure PRP* (cf. Game 1.12, Definitions 1.25 and 1.26). Note that the difference is basically limited oracle access.

Definition 1.39 (PRP Semantic Security Advantage)**.** Let $E : K \times X \to Y$ be a PRP, $\mathcal{E}_b = \{E_b : X^2 \to Y \mid E_b(x_0, x_1) = E(k, x_b), k \in K\}$ for $b = 0, 1$, and A an adversary. We define *PRP semantic security advantage* of adversary A as

$$\mathrm{Adv}_{\mathrm{PRPSS}}(A, E) = \mathrm{Adv}^{(1)}(A, \mathcal{E}_0, \mathcal{E}_1).$$

Definition 1.40 (Semantically Secure PRP)**.** Let $E : K \times X \to Y$ be a PRP. Then it is called *semantically secure PRP* if, for *any* adversary A, the PRP semantic security advantage $\mathrm{Adv}_{\mathrm{PRPSS}}(A, E)$ is negligible.

The following theorem provides a connection of Games 1.31 and 1.38. It claims that, given a PRP E winning Game 1.31, i.e., E is indistinguishable from a truly random bijection, E wins Game 1.38 as well, i.e., no adversary can distinguish encryption of m_0 from encryption of m_1.

Theorem 1.41. Let $E : K \times X \to Y$ be a secure PRP. Then it is also a semantically secure PRP.

Proof. We show that for every PRP semantic security adversary A, there exist PRP security adversaries B_0, B_1 such that

$$\mathrm{Adv}_{\mathrm{PRPSS}}(A, E) \leq \mathrm{Adv}_{\mathrm{PRP}}(B_0, E) + \mathrm{Adv}_{\mathrm{PRP}}(B_1, E). \tag{1.2}$$

The claim then easily follows – if E is a secure PRP, i.e., $\mathrm{Adv}_{\mathrm{PRP}}(B_0, E)+\mathrm{Adv}_{\mathrm{PRP}}(B_1, E)$ is negligible, then also $\mathrm{Adv}_{\mathrm{PRPSS}}(A, E)$ is negligible. Hence E is a semantically secure PRP.

Let us first denote W_b the event when adversary A answers 1 while the true value of the PRP semantic security game is b, for $b = 0, 1$. In this notation, it holds

$$\mathrm{Adv}_{\mathrm{PRPSS}}(A, E) = |\Pr(W_1) - \Pr(W_0)|. \tag{1.3}$$

Let us further denote R_b the same event with only difference – the PRP semantic security game uses a truly random bijection instead. It holds

$$|\Pr(R_1) - \Pr(R_0)| = 0, \tag{1.4}$$

since $b \xleftarrow{R} \{0, 1\}$ and both R_b provide a truly random bijection independent from b.

Now we construct adversaries B_0, B_1 distinguishing PRPs from truly random ones while we make use of adversary A. Let us construct B_0 first. We ask A to provide m_0 and m_1, then we feed the PRP oracle with m_0 obtaining c as a result. Note that either $c = E(k, m_0)$ or $c = F(m_0)$ according to b, where F is a truly random bijection. We provide this c to adversary A, and output what A outputs. The advantage of adversary B_0 is

$$\mathrm{Adv}_{\mathrm{PRP}}(B_0, E) = |\Pr(W_0) - \Pr(R_0)|\,. \tag{1.5}$$

We construct B_1 in a similar manner (i.e., choosing m_1 instead) and get

$$\mathrm{Adv}_{\mathrm{PRP}}(B_1, E) = |\Pr(W_1) - \Pr(R_1)|\,. \tag{1.6}$$

Combining Equations 1.3, 1.4, 1.5 and 1.6, we get

$$\begin{aligned}\mathrm{Adv}_{\mathrm{PRPSS}}(A, E) &= |\Pr(W_1) - \Pr(W_0)| \leq \\ &\leq |\Pr(W_1) - \Pr(R_1)| + |\Pr(W_0) - \Pr(R_0)| = \\ &= \mathrm{Adv}_{\mathrm{PRP}}(B_1, E) + \mathrm{Adv}_{\mathrm{PRP}}(B_0, E),\end{aligned}$$

which finally proves Equation 1.2, hence the theorem, too □

Note 1.42. The previous theorem does not work the other way around, i.e., a semantically secure PRP does not need to be a secure PRP. Indeed, let $E : K \times X \to Y$ be a secure PRP, and let us define $E'(k, x) = E(k, x)\|1$, where $\|$ stands for string concatenation. E' is obviously semantically secure (it remains hard to distinguish encryptions of distinct plaintexts), but it can be easily distinguished from a truly random bijection, since it always ends with 1. The advantage of such adversary would be $\mathrm{Adv}_{\mathrm{PRP}}(A, E') = {}^1\!/_2$ which is clearly non-negligible.

The previous theorem is especially important for the idea behind its proof, which is used in many other proofs related to provable-security. In this proof, we constructed adversaries B_0, B_1, which internally employed adversary A, and then we used advantage of A to estimate advantages of B_0 and B_1.

1.1.6 Provable vs. Real-World Security

Provable security is one aspect, the other is practical construction of a cipher. Note that there is no provably secure block cipher known.

If there were, it would actually imply that $\mathsf{P} \subsetneq \mathsf{NP}$, which is an open millennium problem. Indeed, if $\mathsf{P} = \mathsf{NP}$ and E were a PRP, then the adversary could, given certain (polynomial) amount of plaintext-ciphertext pairs (m_i, c_i), non-deterministically find a key k, such that $c_i = E(k, m_i)$ for all i. Then she would simply answer "truly random" if no such k were found, otherwise she says "pseudorandom". Here, the "truly random" answer is answered with certainty, while the "pseudorandom" answer is just a conjecture, but it has sufficiently large probability.

Instead, real-world block ciphers are *believed* to be secure [38] based on a simple fact: no known attack is far better than exhaustive search. One such block cipher is called *Advanced Encryption Standard* (AES) and will be of top interest in this thesis. AES will be described in detail in Section 1.3.

1.1.7 From Black-Box to White-Box Attack Context

The other central topic of this thesis is *white-box attack context*. In order to explain what it is, let us begin with *black-box attack context*.

Black-Box

First of all, as stated in Note 1.1, we need to address attacker's abilities – the threat model. In the black-box model, the attacker is granted an access to an oracle evaluating given encryption algorithm E with a random secret key k, while the (primary) goal of the attacker is to recover the key. The attacker has absolutely no information but inputs and respective outputs, she cannot observe any internal routines of the encryption procedure, hence *black-box*.

Note 1.43. There could be many other goals than just key recovery, e.g., guessing the ciphertext in advance, or decrypting certain/given ciphertext with non-negligible probability and so on. Actually whatever one can imagine that would be in principle possible with a PRP, but not with a truly random bijection.

Classical ciphers are designed under the assumption of black-box attack context, therefore their implementation must be extremely careful. The attacker could for example measure the time of encryption and exploit this information, which was not considered during the design of the cipher.

Gray-Box

Gray-box attack context is closer to reality, since we do not encrypt using oracles, but rather using some physical device, which runs a specific implementation of given cipher. In the gray-box model, cipher is thus supposed to be implemented and run, while the attacker can observe some information related to cipher's intermediate results as well.

Such information leaks can be modelled as a set of distributions conditioned by intermediate results and are referred to as *leakage model*. There is an area of theoretical cryptography studying leakages – *Leakage Resilient Cryptography*. A practical attack exploiting such leaks will be described in Chapter 2 in Section 2.1.

White-Box

Finally, there is the most extreme case where the attacker has full control over the execution environment – the *white-box attack context* (WBAC). Full control means that the attacker can step the algorithm and directly observe and alter intermediate results in the memory, or even skip, alter and insert instructions. Clearly, the key cannot be kept in its original form, it must be somehow hidden, but still usable for encryption.

1.2 White-Box Cryptography

As we already mentioned, classical ciphers are designed with respect to the black-box attack context, and even under this assumption, it has not been theoretically shown that there exists a secure one; see Section 1.1.6. We also presented gray-box and white-box attack contexts in Section 1.1.7 with a short reasoning of the gray-box model.

Now let us have a closer look at WBAC. As we outlined previously, the attacker is very powerful. And even though she is so powerful, she shall not be able to, besides recovering the key, turn an encryption procedure into decryption one and vice versa.

Let us now justify why we are introducing so extreme case as WBAC. Probably the most intuitive motivation stems from the situation, where a malicious software "lives" in a device together with some software, which is intended to use confidential cryptographic material. For instance, we could use third-party keys (especially in DRM[3]). Note that another motivation will be given after we introduce required concepts (in Note 2.7).

In many use cases, public key cryptography appears to work as well. However, public key (asymmetric) ciphers are far slower than symmetric ones, hence useless in many scenarios. Note that there is no known white-box variant of a public key cipher.

There were several attempts to introduce a "secure" white-box variant of a known cipher. However, most of today's commercial WBAC resistant solutions rather disobey the Kerckhoffs' principle, and rely on security by obscurity and tons of software obfuscation and anti-reverse engineering techniques.

1.2.1 Program Obfuscation

The field of white-box cryptography is closely related to program obfuscation, which was pioneered by Barak et al. [7]. Let us first define what a program obfuscator is, then we give some consequences and present a very interesting theoretical result of their paper.

[3]Digital Rights Management.

Program Obfuscator

Informally, a *program obfuscator* inputs a description of an algorithm (e.g., an input of universal Turing machine or a C code), and transforms it into a functionally equivalent obfuscated algorithm description. Such an obfuscated program should prevent any effective adversary from learning anything she could not learn if she only had an oracle access to the program.

Definition 1.44 (Program Obfuscator; taken from [7], simplified)**.** *Program obfuscator* is an effective (possibly randomized) algorithm, denoted by O, if the following three conditions hold:

1. *functionality:* for every encoding C of an algorithm A_C, $O(C)$ encodes an algorithm that outputs the same as A_C on each input,
2. *polynomial slowdown:* for every encoding C of an algorithm A_C, the length and the running time of $O(C)$ is at most polynomially larger than that of C,
3. *virtual black-box property:* for every effective algorithm B, there exists an effective adversary A such that for every encoding C of an algorithm A_C it holds that

$$\left|\Pr\Big(B(O(C)) = 1\Big) - \Pr\Big(A(\mathcal{O}_{A_C}) = 1\Big)\right|$$

   is negligible in terms of $|C|$.

In other words, the virtual black-box property states that there is no effective algorithm that would be able to gain more information from an obfuscated code than any adversary with an oracle access only, hence black-box property.

Consequences in Cryptography

Besides turning a cipher into a WBAC resistant variant and several applications in software protection, a program obfuscator would allow several other exotic cryptographic constructions as well. As an example, let us mention Fully Homomorphic Encryption (also called *the holy grail of cryptography*, introduced in 1978 by Rivest et al. [50]) or transformation of a private key encryption scheme into a public key one.

Main Result of Barak et al.

However, the main result of Barak et al. basically claims that there is no algorithm that would meet the criteria from Definition 1.44, i.e., *no generic program obfuscator exists.* Although this result might sound disappointedly, it originally claims that there exists *certain class* of algorithms which are unobfuscable. Hence there is no evidence that *any specific* algorithm is unobfuscable. This gives us the hope that there could be some algorithm implementing a block cipher which is not in this class.

Going Beyond

However, the requirements on program obfuscator might be too strong (already suggested by Barak et al.). Indeed, especially in the white-box cryptography context, we can weaken the virtual black-box property, since we probably do not care about any information that does not effectively lead to key recovery, even though we would not learn this information having an oracle access only.

Note that since then, there has been a lot of research using various assumptions, sometimes surprisingly with positive results, too. Most interesting results can be found in [6, 13].

1.2.2 Cat And Mouse Game

It has been conjectured that each white-box resistant implementations gets – sooner or later – broken.

Conjecture 1.45 (Chow et al. [16])**.** No perfect long-term defense against white-box attacks exists.

Therefore it reminds a cat and mouse game. In this thesis, we present both – a white-box resistant implementation and actually two attacks against it. Note that there are another examples from the past, e.g., [15] broken by [26].

1.3 Advanced Encryption Standard

In this section, we present a currently standardized block cipher. Let us begin with how it originated. In 1997, Curtin and Dolske were the first to publicly announce [17] that they have cracked Data Encryption Standard (DES, [46]) – a symmetric block cipher that had been in use for 20 years. In the same year, the National Institute of Standards and Technology (NIST) initiated a selection process for a new encryption standard. During the process, DES was partially replaced with Triple-DES described in the ANSI draft X9.52 [4].

In 1998, NIST chose 15 candidates and asked cryptology community for help with analyzing the candidates. After a careful analysis, NIST proposed Rijndael[4] [18] as the new standard called *Advanced Encryption Standard* (AES) [47] on November 26, 2001. The standard specifies the Rijndael algorithm with block-length of 128 bits and key-lengths of 128, 192 and 256 bits. Note that the original Rijndael algorithm [19] describes more block- and key-length variants. On the other hand, we will only consider its 128-128-bit variant in this thesis, unless stated otherwise.

Today, almost 15 years later, AES is used worldwide and, despite extensive analysis, it is still believed to have considerable security margin.

[4]From authors' names: Vincent Rijmen and Joan Daemen.

1.3.1 AES Building Blocks

AES encryption[5] consists of 10 rounds, each round (except for the last one) consists of 4 different operations, which will be described later. All of these operations work with 128-bit words, let us call them *states*. Before we start, let us present a couple of different state representations.

First, we split its 128 bits into 16 bytes denoted by $A_{0,0}$, $A_{1,0}$, $A_{2,0}$, $A_{3,0}$, $A_{0,1}$, ..., $A_{3,1}$, ..., $A_{3,3}$, respectively, obtaining a 4×4 array, where $A_{i,j}$ stands for an element in i-th row and j-th column.

Let us further denote $F = \mathrm{GF}(2)[x]/x^8+x^4+x^3+x+1$, which is a field of polynomials with integral coefficients reduced modulo 2 (i.e., 0, 1), these polynomials are further reduced modulo $x^8 + x^4 + x^3 + x + 1$. This field has 2^8 elements and is usually referred to as *Rijndael's field*. In the context of AES, it might be denoted also by $\mathrm{GF}(2^8)$.

Each byte $A = a_7a_6 \dots a_0$, $a_i \in \mathrm{GF}(2)$, is then considered as an element of F simply as $a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$. Note that byte may also be written in hexadecimal notation, and still considered as an element of F. See Figure 1.2 with a few of possible AES state representations.

$$01010001\,11010110 \dots \sim \texttt{51 d6} \dots \sim$$

$x^6 + x^4 + 1$	...	...	...
$x^7 + x^6 + x^4 + x^2 + x$	...	...	...
...	...	...	...
...	...	...	...

Figure 1.2: A few of possible AES state representations.

Now let us move on to definitions of the 4 operations.

SubBytes

`SubBytes` operation, often referred to as *SBox*, will be denoted by S. It is defined byte-wise, i.e., $S : F \to F$. For $A \in F$, let A' stand for

- A^{-1}, if $A \neq 0$,
- 0, if $A = 0$.

Note 1.46. Even though A' is not a proper field inverse, it is often referred to as Rijndael inverse of A. Note that Rijndael inverse (taken as mapping) is invertible.

Before we define `SubBytes`, we have to emphasize that the following multiplication is *not* performed in F, but in the ring of binary polynomials $\mathrm{GF}(2)[x]$ instead. `SubBytes`

[5] Decryption will be treated separately at the end of this section.

is defined as follows:

$$S(A) = (x^4 + x^3 + x^2 + x + 1) \cdot A' + x^6 + x^5 + x + 1 \pmod{x^8 + 1}. \tag{1.7}$$

Note 1.47. Although x^8+1 is not irreducible (it holds $x^8+1 = (x+1)^8$), $x^4+x^3+x^2+x+1$ is coprime with x^8+1, therefore it has an inverse modulo x^8+1. It follows that `SubBytes` is an invertible bijection.

`SubBytes` is the only confusion element in AES – indeed, all the other operations will be linear (i.e., from a limited subset of mappings; see Principle 1.28). Since `SubBytes` is the only source of nonlinearity in AES, it shall have excellent nonlinearity properties. This was the major weakness of DES, its predecessor. A practical linear attack against DES was introduced by Matsui [37] in 1993 and analyzed in detail and successfully performed by Junod in 2001 [27].

Remark 1.48. Since `SubBytes` only has 256 possible inputs, it is usually precomputed and implemented as a lookup table, obviously for performance reasons.

The reason why `SubBytes` was not introduced the other way around (i.e., as a lookup table with good nonlinearity properties) is the simple algebraic form of S, which enables mathematical analysis as well as statistical.

Remark 1.49. `SubBytes` is actually an affine mapping (in the vector space F over GF(2)) of Rijndael inverse of its input.

ShiftRows

`ShiftRows` operation is very simple – it only cyclically shifts rows of the state array. It moves i-th row by i positions to the left where rows are numbered from 0; see Figure 1.3. `ShiftRows` is obviously invertible.

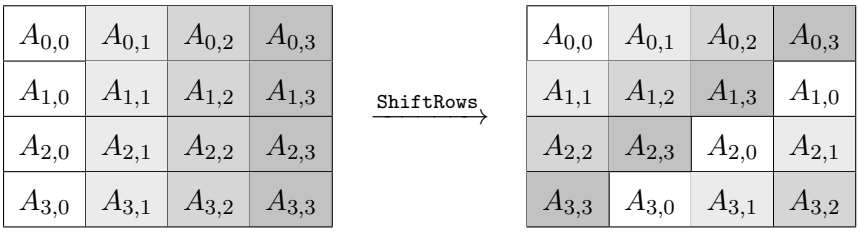

Figure 1.3: `ShiftRows` operation on a state array.

The design motivation for `ShiftRows` was to introduce inter-column diffusion. Note that `ShiftRows` does not diffuse bytes internally – the following operation is here for this purpose.

MixColumns

`MixColumns` operates on the state array in a column-wise manner. Let us pick a column from the state array and denote its elements by A_0, A_1, A_2, A_3, respectively. `MixColumns`

considers it as a polynomial $a(z) = A_0 + A_1 z + A_2 z^2 + A_3 z^3$ – an element of $F[z]$. Let $b(z)$ denote the output of `MixColumns`, then it is defined as

$$b(z) = c(z) \cdot a(z) \pmod{z^4 + 1} \tag{1.8}$$

where $c(z) = \texttt{03} \cdot z^3 + \texttt{01} \cdot z^2 + \texttt{01} \cdot z^2 + \texttt{02}$. Note that here we used hexadecimal notation for elements of F, which can also be viewed as polynomials. To avoid confusion, we used z instead of x.

Note that $c(z)$ is coprime with z^4+1, therefore invertible modulo z^4+1. It follows that `MixColumns` is invertible. The goal of `MixColumns` is to introduce inter-byte diffusion within a column.

Later we will make use of its different, linear algebra representation. Let us denote $b(z) = B_0 + B_1 z + B_2 z^2 + B_3 z^3$. Modular multiplication in Equation 1.8, which is performed in $F[z]$, can be rewritten into a matrix multiplication back in F, indeed

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} = \begin{pmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix}. \tag{1.9}$$

AddRoundKey

Given a 128-bit round key, `AddRoundKey` operation simply bitwise XORs it on the state. The round key is derived from the encryption key using `KeySchedule` routine which follows.

`AddRoundKey` is the only step that employs keying material. Note that it is its own inverse.

KeySchedule

`KeySchedule` is a routine which expands 128-bit encryption key into 11-times longer `ExpandedKey`. 128-bit blocks of `ExpandedKey` serve as round keys. Their position within `ExpandedKey` is indexed from 0.

Note 1.50. The most important consequence for us is that `ExpandedKey` begins with plain encryption key.

We omit further details here, because `KeySchedule` is specified rather in an algorithmic than algebraic manner; see [19, pp.43-45] for detail.

1.3.2 AES Algorithm Description

AES, as already stated, consists of 10 rounds of 4 operations except for the last one where `MixColumns` is missing; see the following algorithm.

Algorithm 1.1. Given a 128-bit plaintext and key, return its AES encryption.

```
1: function AES_Encryption(Plaintext, Key)
2:     ExpandedKey ← KeySchedule(Key)
3:     State ← Plaintext
4:     AddRoundKey(State, ExpandedKey[0])
5:     for Round = 1 → 9 do
6:         SubBytes(State)
7:         ShiftRows(State)
8:         MixColumns(State)
9:         AddRoundKey(State, ExpandedKey[Round])
10:    end for
11:    SubBytes(State)
12:    ShiftRows(State)
13:    AddRoundKey(State, ExpandedKey[10])
14:    return State
15: end function
```

Note 1.51. Since we have commented on every step about how it can be inverted, AES encryption algorithm is hence also invertible. We get AES decryption algorithm by executing inverse counterparts of AES encryption steps in reverse order.

1.3.3 AES Implementation Note

As stated in Remark 1.48, `ShiftRows` can be implemented as a lookup table for performance reasons. Unlike `SubBytes`, `MixColumns` operates on a substantially larger domain, therefore it cannot be directly implemented as a lookup table. But we can utilize its matrix representation, see Equation 1.9. Let us denote

$$\mathbb{C} = \begin{pmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{pmatrix},$$

we can rewrite the equation to the form

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} = \mathbb{C} \begin{pmatrix} A_0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \mathbb{C} \begin{pmatrix} 0 \\ A_1 \\ 0 \\ 0 \end{pmatrix} + \mathbb{C} \begin{pmatrix} 0 \\ 0 \\ A_2 \\ 0 \end{pmatrix} + \mathbb{C} \begin{pmatrix} 0 \\ 0 \\ 0 \\ A_3 \end{pmatrix} =$$
$$= \mathbb{C}_0 \cdot A_0 + \mathbb{C}_1 \cdot A_1 + \mathbb{C}_2 \cdot A_2 + \mathbb{C}_3 \cdot A_3,$$

where $\mathbb{C}_i$ denotes i-th column of $\mathbb{C}$.

Note that each addend only depends on a single byte, i.e., there are just 256 possible input values for each. Therefore `MixColumns` can be implemented as a four-tuple of lookup tables, the result is then simply a bitwise XOR of their 4-byte output values.

This was just to give an idea about how AES operations can be turned into lookup tables. These can be further improved, see [19, Chapter 4] for detail.

1.4 White-Box AES

In this section, we describe the White-Box AES (WBAES) as introduced by Chow et al. [16] and revisited by Muir [41] (which we highly recommend). Note that it has already been broken by an interesting algebraic attack by Billet et al. [10], which will be mentioned further in Section 1.4.5.

The core idea of WBAES is based on the lookup table implementation as outlined in Section 1.3.3, these tables are further improved in three ways:

1. the implementation is turned into a *fully* table representation, i.e., the key is incorporated into tables as well,
2. tables are wrapped around with appropriately self-vanishing random bijections,
3. the boundary is extended into the containing application.

Note that Chow et al. introduced also the White-Box DES [15], but it has been broken by Jacob et al. [26] much earlier than WBAES, because it was lacking feature 3 from the previous list.

Notation 1.52. Since WBAES will use lookup tables heavily, let us introduce a simplifying notation: a table will be told to have dimensions $m \times n$ if it has m-bit input values and n-bit output values. Note that the real size (i.e., memory consumption) of such table is $2^m \cdot n$ bits, since we need 2^m entries of size n to treat all possible m-bit input values.

1.4.1 Encodings

Let us first introduce some notion for the feature 2 from the previous list as defined by Chow et al.

Definition 1.53 (Encoding)**.** Let X be a transformation from m bits to n bits. Choose an m-bit bijection F and an n-bit bijection G. Call $X' = G \circ X \circ F^{-1}$ an *encoded* version of X. F is an *input encoding* and G is an *output encoding*.

Note 1.54. Given a fixed key, AES itself is a 128-bit bijection, which can obviously hardly be implemented as a lookup table, because its size would be $2^{128} \times 128$ bits. On the other hand, this would be black-box equivalent – we only know the input/output behavior.

Hence we will rather only consider a special kind of bijections – we will use series of smaller bijections and concatenate their outputs, a definition follows.

Definition 1.55 (Concatenated Encoding)**.** Let F_i be n_i-bit bijections for $i = 1, \ldots, k$ and let $n = n_1 + \ldots + n_k$. The *function concatenation* $F_1 \| F_2 \| \ldots \| F_k$ is the n-bit bijection F such that $F(b) = F_1(b_1, \ldots, b_{n_1}) \| F_2(b_{n_1+1}, \ldots, b_{n_1+n_2}) \| \ldots \| F_k(b_{n_1+\ldots+n_{k-1}+1}, \ldots, b_n)$ for any n-bit word $b = b_1 \| b_2 \| \ldots \| b_n$, where $\|$ denotes word concatenation (might be omitted). Clearly $F^{-1} = F_1^{-1} \| F_2^{-1} \| \ldots \| F_k^{-1}$.

The following definition introduces the self-vanishing property of the bijections.

Definition 1.56 (Networked Encoding)**.** A *networked encoding* for computing $Y \circ X$ (i.e., transformation X followed by transformation Y) is an encoding of the form

$$Y' \circ X' = (H \circ Y \circ G^{-1}) \circ (G \circ X \circ F^{-1}) = H \circ (Y \circ X) \circ F^{-1}$$

where F, G, H are bijections of appropriate size, F and H are *external encodings* (input and output encoding, respectively) and G is an *internal encoding*.

Note that, given only tables for Y' and X', G can be totally forgotten and we are still able to compute $Y \circ X$, provided that we know the input and output encoding. Such a networked encoding can be obviously applied to much longer composition of transformations and we still only need to know the input and output encoding.

1.4.2 Reordered AES

In order to begin with fully table representation, we need to reorder AES operations, so that the resulting tables can be easily composed. Let us first have a brief look at AES – Algorithm 1.1.

We "move" the `for` cycle on Line 5 one row upwards with appropriate shift of indexes in `ExpandedKey`. Then we switch `ShiftRows` and `SubBytes` without any side-effect since `SubBytes` operates byte-wise. Finally, we switch `ShiftRows` and `AddRoundKey`, while we moreover have to apply `ShiftRows` on `ExpandedKey` in order to get an equivalent algorithm. The modified version looks as follows:

Algorithm 1.2.

```
1: function Reordered_AES_Encryption(Plaintext, Key)
2:     ExpandedKey ← KeySchedule(Key)
3:     State ← Plaintext
4:     for Round = 0 → 8 do
5:         ShiftRows(State)
6:         AddRoundKey(State, ShiftRows(ExpandedKey[Round]))
7:         SubBytes(State)
8:         MixColumns(State)
9:     end for
10:    ShiftRows(State)
11:    AddRoundKey(State, ShiftRows(ExpandedKey[9]))
12:    SubBytes(State)
13:    AddRoundKey(State, ExpandedKey[10])
14:    return State
15: end function
```

1.4.3 Fully Table Representation

As outlined earlier, WBAES construction turns all AES operations into table lookups only. Moreover, we compose certain tables and wrap them all around with internal encodings in the fashion of Definition 1.56. WBAES generator thus inputs an AES key and a random seed, and outputs key-dependent tables which serve as a WBAES instance.

Note 1.57. Some tables of a given type (e.g., `SubBytes` table) might be equal across or within rounds, but we will consider every possible instance of each table, since we will give them later distinct encodings.

T-Boxes

First, we compose `AddRoundKey` with the subsequent `SubBytes` step, yielding an operation referred to as `TBox`, also denoted by $T^r_{i,j}$ for r-th round and (i,j)-th position in the state array. We get

$$T^r_{i,j}(x) = S(x + k^r_{sr(i,j)}), \qquad i,j = 0,\ldots,3,\ r = 0,\ldots,8,$$

where $k^r_{sr(i,j)}$ stands for the `ExpandedKey`'s byte in the r-th round at the position (i,j) shifted by `ShiftRows` (here denoted by $sr(i,j)$). The last round is treated individually as

$$T^9_{i,j}(x) = S(x + k^9_{sr(i,j)}) + k^{10}_{i,j}, \qquad i,j = 0,\ldots,3. \tag{1.10}$$

Single `TBox` can be implemented as an 8×8 lookup table[6], and there are $16 \cdot 10 = 160$ of them.

MixColumns

`MixColumns` (here also denoted by `MC`), acting on a single column, can be turned into four 8×32 lookup tables followed by three XOR operations as outlined in Section 1.3.3. Since there are 9 uses of `MixColumns`, each acts on 4 columns, and each such requires 4 tables, `MixColumns` requires 144 tables altogether.

The difference from the classical implementation is that the subsequent XOR operations must be turned into lookup tables as well. Since the input size of such XOR operation would be $2 \cdot 32$ bits, we need to split inputs into smaller 4-bit segments and treat them separately in the fashion of concatenated encoding (Definition 1.55).

The size of such a single XOR table is then $2 \cdot 4 \times 4$, and we need to perform $32/4 \cdot 3 = 24$ such lookups per one column and per one `MixColumns`. It follows that we need altogether $24 \cdot 4 \cdot 9 = 864$ of such XOR tables (according to Note 1.57).

ShiftRows

`ShiftRows` does not need table representation, since it only moves bytes within the state.

[6] Reminder of Notation 1.52.

Table Composition

Note that `TBox` is implemented as an 8×8 table, which is followed by an 8×32 table of `MixColumns` in rounds $0, \ldots, 8$. Therefore it is reasonable to compose those appropriately subsequent tables together – we save both space and time.

It follows that altogether we have 144 composed $\texttt{TBox} \circ \texttt{MixColumns}$ tables, 864 XOR tables and 16 final round `TBox` tables.

1.4.4 WBAES Construction

So far, we have turned all AES operations into lookup tables only. Next we will incorporate encodings as described in Section 1.4.1, these encodings will play the role of a confusion feature. Then introduce a diffusion feature called *Mixing Bijection*, and finally compose certain tables together.

Inserting Encodings

Note that the 32-bit outputs of `MixColumns` are first split into 4-bit words, and then two independent 4-bit words are XORed together, yielding a new 4-bit word. Hence we are limited to use (only) 4-bit internal encodings, here denoted by `Enc`, on both sides of each XOR. This is the reason why *input mixing bijections* will be introduced – they will diffuse two neighboring 4-bit blocks together.

Mixing Bijections

Mixing bijection (denoted by `MB`) is a random column-wise (i.e., 32-bit) linear bijection that is to be inserted after `MC` and inverted in a separate step (this needs to be done before applying encodings). Since it inputs the output of `MC`, they can be composed together, yielding $\texttt{MB} \circ \texttt{MC}$. Note that table representation of both $\texttt{MB} \circ \texttt{MC}$ and $\texttt{MB}^{-1}$ can be created in very the same manner as previously for `MC`.

Note 1.58. Chow et al. claim that all aligned 4×4 submatrices of `MB` must have full rank. This requirement is related to using 4-bit encodings introduced previously.

Input Mixing Bijections

Input mixing bijection (`IMB`) is a byte-wise variant of mixing bijection. It provides diffusion after inverting mixing bijection and before entering next `TBox`. Since it is a linear mapping, it can be composed with the inverted mixing bijection, and since it is a byte-wise mapping, it can be composed with `TBox` as well.

The only remaining operation – `ShiftRows` – will be treated separately later.

Sketch of Single Round

Let us demonstrate previous ideas in a single sketch of the flow through one inner round (i.e., not the first round nor the last) showing all operations, table compositions, mixing

bijections and encodings. Tables are given their names (e.g., Type II) as introduced by Chow et al., and "plain" stands for any intermediate result of the original AES (i.e., without being hidden by another mapping).

$$\ldots \to \underbrace{\texttt{Enc} \to \texttt{IMB}^{-1} \xrightarrow{\text{plain}} \texttt{TBox} \xrightarrow{\text{plain}} \texttt{MB} \circ \texttt{MC} \to \texttt{Enc}^{-1}}_{\text{Type II}} \to \Big(\underbrace{\texttt{Enc} \to \oplus \to \texttt{Enc}^{-1}}_{\text{Type IV}}\Big)^2 \to$$

$$\to \underbrace{\texttt{Enc} \to \texttt{IMB} \circ \texttt{MB}^{-1} \to \texttt{Enc}^{-1}}_{\text{Type III}} \to \Big(\underbrace{\texttt{Enc} \to \oplus \to \texttt{Enc}^{-1}}_{\text{Type IV}}\Big)^2 \to \ldots \tag{1.11}$$

Interested reader is referred to Miur's tutorial [41], where they give large and well-arranged figures depicting the flow in detail, including I/O bitsizes or depicting incorporation of `ShiftRows` operation. Note that I/O bitsizes differ across WBAES mappings, and Equation 1.11 does not provide that information.

ShiftRows

`ShiftRows` only moves bytes within the state. Indeed, at the time of applying `ShiftRows` (i.e., before entering Type II table), only encoding and input mixing bijection are applied, both operate byte-wise.

External Encodings

So far, we have only treated internal rounds, i.e., rounds $1, \ldots, 8$ in reordered AES (see Algorithm 1.2). There is one more issue with tables on the very input and on the very output – these can handle either non-encoded, or appropriately encoded data. Chow et al. introduced *external encodings* for the latter and gave it a reasoning: as another level of obfuscation, encoded plaintexts and ciphertexts are used instead of raw ones; encoding can then be hidden within a larger containing application or implemented separately.

Let E_k stand for AES encryption using key k, then encryption using external encodings can be written as

$$E'_k = G \circ E_k \circ F^{-1}, \tag{1.12}$$

where G, F stands for input and output external encoding, respectively. These are basically 128-bit bijections. Clearly, the input plaintext must first be encoded with F, then encrypted with encoded variant E'_k, and finally decoded with G^{-1}, in order to get an equivalent of AES encryption.

Chow et al. suggest to use 128-bit linear mappings as external encodings. These can be further composed with corresponding input mixing bijections of the first and the last round, respectively, preserving linearity.

Such external encodings can be obviously implemented as lookup tables in a manner similar to `MixColumns`, here using 16 tables[7] of size 8×128 followed by appropriately encoded XOR tables. Note that there are 4 levels of XOR tables now, since we have

[7] A new, larger type of tables is needed.

16 vectors to be XORed together. Also note that the very input and output are *not* encoded with `Enc`.

See Equations 1.13 and 1.14 for a sketch of the flow through input and output external encoding, respectively. Note that external encoding can be trivially turned off by setting E and F equal to identity.

$$\xrightarrow{\text{input}} \underbrace{\texttt{IMB} \circ G \to \texttt{Enc}^{-1}}_{\text{Type I}} \to \left(\underbrace{\texttt{Enc} \to \oplus \to \texttt{Enc}^{-1}}_{\text{Type IV}}\right)^4 \to \dots \tag{1.13}$$

$$\dots \to \underbrace{\texttt{Enc} \to F^{-1} \circ \texttt{IMB}^{-1} \to \texttt{Enc}^{-1}}_{\text{Type I}} \to \left(\underbrace{\texttt{Enc} \to \oplus \to \texttt{Enc}^{-1}}_{\text{Type IV}}\right)^3 \to \underbrace{\texttt{Enc} \to \oplus}_{\text{Type IV}} \xrightarrow{\text{output}} \tag{1.14}$$

1.4.5 Known Attacks & Enhancements

As outlined in Section 1.2.2, sooner or later after introduction of a new white-box implementation of a cipher somebody usually comes up with an effective attack. This is also the case of WBAES introduced by Chow et al. in 2002 as well, it was broken by Billet et al. [10] two years later. The attack is called *BGE attack* – it is an abbreviation of authors' names.

BGE Attack

BGE attack is a pure algebraic attack, therefore it requires access to all tables implementing WBAES.

Note 1.59. In a real-world scenario, we first need to reverse-engineer the provided binary code in order to access the tables. This can make any analysis really painful – such white-box cryptography programs are often further protected against reverse engineering effort with various techniques.

We will not cover the BGE attack in detail here, it is out of the scope of this thesis, but it motivated further research. On the one hand, the BGE attack has been adapted for a broader class of ciphers by Michiels et al. [39], on the other hand, novel white-box approaches have been introduced. Among others (e.g., [40, 55]), we focus on Karroumi's contribution [28]. Even though it has been shown by Klinec [30] to be equivalent to the original WBAES by Chow et al., it will later support the origin of our contribution.

Karroumi's Approach with Dual Ciphers

Karroumi noticed that the BGE attack exploits the knowledge of specific AES constants, for instance coefficients inside `MixColumns` (see Equation 1.9) or, more importantly, coefficients of affine transformation inside `SubBytes` (see Equation 1.7 and Remark 1.49). In 2002, Barkan et al. [8] came up with the idea of changing these coefficients in an

appropriate way and Karroumi suggested to use such modified AES as a basis for an upgraded WBAES. In general, such a modified cipher is referred to as a *dual cipher*.

Originally there were 240 dual AES ciphers by Barkan et al., the list has been later extended by Raddum [49] to 9 360 and further by Biryukov et al. [11] to 61 200 dual AES ciphers.

Note that there is a simple relation between the AES and a dual AES: for each dual cipher, there exists a linear mapping Δ that maps a state of the AES on a state of the dual AES and vice versa. Since both plaintext and ciphertext are an AES state at some point, it follows that $P_{dual} = \Delta P$ and $C_{dual} = \Delta C$, moreover the same holds nontrivially for the key, i.e., $K_{dual} = \Delta K$.

Karroumi incorporates $4 \cdot 10$ distinct dual AES'es into WBAES (for each column of each round), while appropriate linear mappings Δ, are simply composed with mixing bijections in each round. Hence the overall structure of WBAES remains unchanged. Karroumi presumes that this modification raises the complexity of the BGE attack from originally 2^{30} to 2^{91}, since the attacker seems to need to loop through all four-tuples of dual AES'es and to run the standard BGE with constants of those dual AES'es.

Intuitively, since the transfer to dual cipher is only done by a linear transformation, and since each original WBAES round is wrapped around with a random linear transformation (mixing bijection), it seems that we can consider each original WBAES round as internally using a dual AES and vice versa. Klinec has shown in [30, Proposition 2] that this is indeed true.

Klinec's Equivalence Result

Here we state Klinec's result, but note that, according to the original proof, it is rather a consequence of a much stronger property, see below.

Proposition 1.60 (Klinec)**.** Karroumi's WBAES scheme can be broken with the BGE attack with the same time complexity as the original WBAES scheme.

Proof. We refer to [30] for the full proof, it is rather long and technical and requires a broader context, too. □

Remark 1.61. The core observation of Klinec's proof is that equations of Karroumi's WBAES can be rewritten to the form of the original Chow's WBAES, i.e., without using dual ciphers at all. Concretely the transformations to dual cipher and operations in dual cipher are shown to be absorbed by surrounding random bijections (namely $\texttt{IMB}^{-1} \circ \texttt{Enc}$ and its inverse, in addition also by `MB` and its inverse), yielding the original Chow's WBAES.

As a consequence of the previous remark, it follows that the BGE attack breaks Karroumi's WBAES with the same time complexity as the original WBAES scheme – simply because both is the same from the attacker's perspective.

Note 1.62. Later in this thesis, we will attack Klinec's implementation [29] of WBAES by Chow et al. We will utilize Remark 1.61 in a slightly different way.

Note that, among others, dual AES allows us to alter coefficients of the affine transformation inside `SubBytes` step. Since we do not recognize which instance of dual AES has been used in WBAES, we can expect every single one, i.e., we can suppose that those SBoxes use *arbitrary invertible affine mapping*.

Chapter 2

Using Side-Channel Attack Tools

In this chapter, we first describe an attack exploiting information leaks assumed in gray-box attack context (see Section 1.1.7). This attack is referred to as *Side-Channel Attack* (SCA) and we show two SCA algorithms, both practically usable.

Next we present a novel approach of Bos et al. [12] utilizing SCA tools in the white-box attack context.

2.1 Side-Channel Attack

Let us assume gray-box attack context. In practice, there are several means of information leakage – for instance power consumption [32], electromagnetic radiation [2, 22, 48], timing [33], or even acoustic [3] and optical [34, 35] emissions may carry some information related to ciphering algorithm's intermediate results or execution. The goal of SCA is to exploit these information leaks and recover some secrets.

SCA was pioneered by Kocher [33] in 1996. His paper was originally concentrating on public key cryptography, but the general idea can be ported to symmetric cryptography as well.

Since then, several techniques have been developed, including Simple Power/EM Analysis (SPA/SEMA), Differential Power/EM Analysis (DPA/DEMA), High-Order DPA/DEMA, Correlation Power Analysis (CPA), Inferential Power Analysis (IPA), partitioning attacks, collision attacks, hidden Markov model, etc.; see [31, Chapters 13-14] for reference.

The success of SCA is based on the following assumptions:

1. the cipher is insecure once certain execution-related information leaks (e.g., intermediate results or runtime),
2. the attacker controls, knows or just can predict certain amount of plaintext inputs, and
3. the attacker has the ability to gain some information related to execution of the cipher.

Let us present this idea on a concrete attack.

2.1.1 Correlation Power Analysis Attack

In this section, we describe the CPA attack (an advanced form of the DPA attack) mounted on an unprotected AES implementation, where the attacker can measure power consumption, referred to as *power trace*; see Figure 2.1 for an example power trace. First, we show which AES intermediate result leads to immediate key recovery, then we describe how the attacker can utilize information related to this intermediate result hidden in power traces to recover the key. Remind Algorithm 1.1 in Section 1.3 for AES construction.

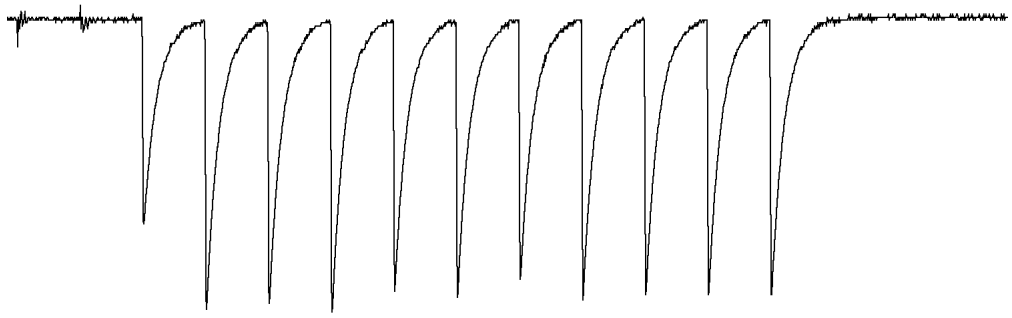

Figure 2.1: SAKURA-G running AES, note its 10 rounds. Data taken from [44].

Vulnerable Intermediate Result

In AES, the very first plaintext-dependent operation is `AddRoundKey` on Line 4 of Algorithm 1.1, which XORs current state with the first block of expanded key. Here the state is equal to input plaintext, see Line 3, and the first block of expanded key is equal to the original AES key, see Note 1.50. `AddRoundKey` is then usually composed with `SubBytes` (the following step), hence yielding

$$b = \texttt{SubBytes}(k \oplus m) \tag{2.1}$$

as the first (128-bit wide) intermediate result, where k stands for the AES key and m for the plaintext. b is then used in the following AES step which is `ShiftRows`, usually composed with `MixColumns`.

Once we would learn b and respective m from a single AES encryption, it would be easy to directly compute the key as $k = \texttt{SubBytes}^{-1}(b) \oplus m$.

Utilizing Power Traces

Yet in this attack scenario, we do not observe intermediate results b directly, we only have power traces. Now let us describe a reasonable model how a certain portion of information about b's leaks into the power trace, and how we can exploit it.

Before entering next stages, the intermediate result b is temporarily stored. It is handled byte-wise, hence its byte-wise Hamming weights (i.e., numbers of ones in binary representation, further denoted by HW) are reflected into the power trace[1]. It means that for each byte of b denoted by b_i, there is a position in the power trace that reflects $HW(b_i)$.

Let us suppose that we have a plenty of such traces which are properly aligned, i.e., a position in the traces corresponds with a fixed position in the AES implementation. Therefore, for each byte of b's, there is a position in the traces, where the power values are correlated with Hamming weights of respective bytes of respective b's.

Since we do not know actual b's, we will assume that the correlation of true values of b's is the highest among others, and among all positions in the traces. Note that $b_i = S(k_i + m_i)$, i.e., b's only depend on respective known plaintexts m and the key k which is constant, therefore we will loop through key guesses and search for globally maximal correlation. And more importantly, note that we can guess the key byte-wise!

Note 2.1. In the CPA, we loop through 256 key guesses for each of 16 bytes, i.e., overall $256 \cdot 16 = 2^{12} = 4096$ possibilities, compare with $256^{16} = 2^{128} \approx 3.4 \cdot 10^{38}$ possibilities of a brute force attack.

Attack

The attack performs as follows: given a byte index $i = 0, \ldots, 15$, we find the maximal correlation[2] (among all keyguesses k_i of i-th byte of the key and all positions in the traces) of $HW(S(k_i + m_i))$ with respective power values. k_i reaching the maximum is then our top candidate. This idea is summarized in the following algorithm.

Algorithm 2.1. Given power traces and respective plaintexts, recover the AES key.

1: **function** CPA_Attack($Traces[N][T], PTs[N]$)
2: **for** $i = 0 \to 15$ **do**
3: $max \leftarrow 0$
4: **for** $KGuess = \texttt{00} \to \texttt{ff}$ **do**
5: $tmpmax \leftarrow \max_{t \in T} \operatorname{Corr}_{n \in N}\Big(Traces[n][t], HW\big(S(KGuess \oplus PTs[n][i])\big)\Big)$
6: **if** $tmpmax > max$ **then**
7: $max \leftarrow tmpmax$
8: $BestKey[i] \leftarrow KGuess$
9: **end if**
10: **end for**
11: **end for**
12: **return** $BestKey$
13: **end function**

[1] Here we assume the so-called *Hamming Weight Leakage Model*.

[2] We can rather use some other empirical meter, since correlation is quite expensive and we only need to find the key, no matter how.

Remark 2.2. We obviously do not compute SBox and Hamming weight at each step, but we rather use a single precomputed table of the composition of $HW \circ S$. Clearly, this table has 256 entries in the range $0, \ldots, 8$.

Note 2.3. It is useful to modify the previous algorithm to output the whole list of key candidates together with their maximal correlation values for each key byte, i.e., the result of Line 5, not only $BestKey$.

This is because if $BestKey$ is not equal to the true key, we still have a good chance to recover the key: since the true key is likely to have a rather high correlation, we loop through key candidates according to their correlation values – in the descending order.

Note that we can verify the correctness of our key guess simply by checking whether the resulting ciphertext equals to the ciphertext we expect.

Note 2.4. It might be also helpful to output positions of maximal correlation for each candidate. This information might be valuable for any further analysis purposes.

There are several solutions implementing the CPA including the open-source ChipWhisperer™ Project [25], which is a toolkit providing a complete toolchain from trace acquisition to key extraction. We derived this algorithm from the ChipWhisperer™documentation.

2.1.2 Bitwise Differential Power Analysis Attack

Let us continue with another attack on an unprotected AES implementation. For now, let us suppose a more powerful attacker, who can probe individual wires of a vulnerable bus and measure its voltage. Let us further suppose that the intermediate results $b = \texttt{SubBytes}(k \oplus m)$ are transferred over this bus.

Therefore such an attacker is not limited to observing values related to the Hamming weight of b, but she can rather measure values related to individual bits of b! The advantage of such attacker is that there suffice much less measurements to successfully recover the key.

Attack

In this attack, we proceed similarly as in the previous one, i.e., we loop through key guesses for each key byte and compute the respective intermediate results b_i; see Algorithm 2.1. But instead of applying Hamming weight and computing correlation, we proceed with each bit individually.

Let us say we are targeting the j-th bit of the i-th byte of b. Based on the actual key guess k_i, we compute the j-th bit of $b_i = S(k_i + m_i)$ for each trace/plaintext pair, and partition the traces based on this bit into two sets, let us denote them S_0 and S_1, respectively.

Note 2.5. Typically, we have several traces (from several probes) for a single plaintext and we do not know which of them contains which bit. Actually, this does not play much of a role – we can serialize them into a single trace and do not care anymore.

If the key guess is correct, then there is a position in the traces, where the values are low in traces in S_0 and high in traces in S_1. We can find this position as the spot of the

highest difference of mean values within S_0 and S_1. If the key guess is not correct, there should be no substantial peak in this difference of means. Hence the true key should maximize the trace-maximal difference of means among other key guesses. An algorithm implementing this attack follows.

Algorithm 2.2. Given voltage traces and respective plaintexts, recover the AES key.

1: **function** BITWISE_DPA_ATTACK($Traces[N][T], PTs[N]$)
2: **for** $i = 0 \to 15$ **do**
3: $max \leftarrow 0$
4: **for** $KGuess = \texttt{00} \to \texttt{ff}$ **do**
5: **for** $j = 0 \to 7$ **do**
6: $S_0 \leftarrow \left\{ Traces[n] \;\middle|\; S(KGuess \oplus PTs[n][i])[j] = 0 \right\}$
7: $S_1 \leftarrow \left\{ Traces[n] \;\middle|\; S(KGuess \oplus PTs[n][i])[j] = 1 \right\}$
8: $tmpmax \leftarrow \max_{t \in T} \left| \underset{Trace \in S_1}{\text{Mean}} (Trace[t]) - \underset{Trace \in S_0}{\text{Mean}} (Trace[t]) \right|$
9: **if** $tmpmax > max$ **then**
10: $max \leftarrow tmpmax$
11: $BestKey[i] \leftarrow KGuess$
12: **end if**
13: **end for**
14: **end for**
15: **end for**
16: **return** $BestKey$
17: **end function**

Note 2.6. We can modify this algorithm in a manner similar to Notes 2.3 and 2.4, and output the full record of maximal differences of means (instead of correlations previously, here it is the result of Line 8) and their positions in traces. Note that in this case we get 8 separate lists for each j, see Line 5 in the previous algorithm. Here j is referred to as the *target bit*. We can also unravel which target bit leaks most substantially, which might be useful for further analysis purposes.

To the best of our knowledge, there is no open-source project implementing this attack. This method has been taken from [53].

2.2 Using SCA Tools in White-Box Attack Context

In the previous section, we presented some algorithms seemingly unrelated to what we have done so far. Indeed, our aim is to attack white-box implementations, where we can fully control the execution environment, thus there is no need for physical measurements. The actual reason emerges from a novel approach introduced recently by Bos et al. [12] – the use of SCA tools in the context of white-box cryptography.

The idea was inspired by Delerablée et al. [20]: as it is usually required, a "perfect" white-box implementation should not provide effectively any other information about the

secret key, but only as much as an access to a black-box. Delerablée et al. observed that such implementation must be resistant to all existing and future side-channel attacks, which led Bos et al. to the idea of exploring this consequence by attacking white-box implementations with SCA tools. And the attack succeeded!

Note 2.7. The previous observation also implies another motivation for white-box cryptography – a white-box attack resistant implementation could be used for a SCA-resistant hardware implementation.

The obvious question is what kind of traces we should use instead of power traces – we use *software traces*. In this context, software trace refers to a record of memory addresses being accessed during program execution, or their contents; sometimes also referred to as the *memory trace*. Hence in this context, differential power analysis might sound strange, since there is no *power*, it will be rather referred to as the *Differential Computation Analysis* (DCA).

Remark 2.8. We would like to especially highlight that this approach avoids reverse engineering – it only exploits leaked runtime information – which is probably the best practical benefit. There is further no need for knowledge of the design, obviously.

In the following sections, we describe the full process of the DCA attack – from trace acquisition, over trace filtering, concluding with the attack itself.

2.2.1 Trace Acquisition

In order to acquire a memory trace, we used Dynamic Binary Instrumentation (DBI) tools. These tools insert additional instructions to the original code of the program at run-time, enabling one to debug or detect memory leaks. The most advanced DBI tools, like Valgrind [42] or PIN [36], include a programmable interface, where one can write tools of own will. Both tools are open-source.

We modified a tool for PIN by Teuwen [53] for 4 use cases. These acquire

1. addresses of memory reads,
2. addresses of memory writes,
3. contents of memory reads, and
4. contents of memory writes,

respectively. Tool No. 4 is obviously most useful for unprotected implementations, since it can directly observe intermediate results, on the other hand, tool No. 1 will be surprisingly used as well. The other tools will not be used in this thesis.

Note 2.9. So far, we have not specified any possible limitations on memory traces. Note that in case we are catching addresses, we can limit our attention to the least significant byte only. Indeed, the rest of the address typically does not contain any relevant information, only the global position within the memory, but the least significant byte carries information about the position within an array representing a WBAES table.

In case of reading memory contents, we can only be interested in 1-byte memory reads/writes, since AES tables are often constructed based on byte-wise representation.

Using the chosen tool, we acquire certain amount of memory traces with different, possibly random, plaintexts on input. We save these plaintexts along corresponding traces, of course.

Note 2.10. In order to attack with SCA tools, traces must be properly aligned. It could happen that compiling C/C++ programs with high levels of optimization would produce misaligned traces – we indeed experienced different lengths of traces.

Therefore we recommend to compile C/C++ programs with different levels of optimization and acquire just a few traces for each level. We check traces' lengths and apply the highest optimization level, where the traces were equally long, and use this optimization level for trace acquisition.

One may wonder whether traces can ever be properly aligned, because then it would require additional effort to align them. According to our experience, traces were well aligned, hence there should be no need for this[3].

Note 2.11. There is another issue with trace acquisition, now it is related to the next step which is trace filtering. It will be highly appreciated that the program uses the very same addresses for instructions that are not related to the plaintext. Later we will try to filter them out, since they do not carry any useful information.

For this reason, it might be helpful to switch off the Address Space Layout Randomization (ASLR) and acquire all traces in a single terminal session.

Having traces properly acquired, let us proceed to trace filtering.

2.2.2 Trace Filtering

Once we have acquired memory traces, we perform two kinds of filtering. We filter trace entries by

- constant value, and
- address and temporal range.

Filtering of Constant Entries

In the first stage, we filter out such entries that are constant across all acquired traces, because they do not carry any information. Indeed, both presented SCA algorithms exploit changes across traces.

Practically, we first create a filter mask based on a small subset of traces, 10 appears to be sufficient. Such a mask carries 0 if all corresponding addresses in the small set of traces are equal, 1 otherwise; see an example in Figure 2.2. This mask is then simply applied to filter all the remaining traces, possibly hundreds or thousands of them. Note that we keep the mask, we will use it for the following filtering method as well.

[3]Later we will introduce some countermeasures against this attack, including attempts to misalign traces. We will mention then another approach how we can treat trace misalignment.

Trace 0	65	de	37	7e	63	30	c0	62	61
Trace 1	65	de	31	7e	23	30	b1	62	61
Trace 2	65	de	1f	7e	d5	30	77	62	61
Trace 3	65	de	97	7e	5c	30	c4	62	61
Mask	0	0	1	0	1	0	1	0	0

Figure 2.2: An example of a mask used for trace filtering.

Note 2.12. Filtering of constant entries typically decreases the size of the traces by tens of percent. If we did not observe any or extremely small decrease, it might have happened that the traces were not properly aligned. In such case, it might be difficult to align them. Fortunately, we did not experience this problem (we gave some advices how to avoid this in Note 2.11).

Address And Temporal Filtering

In the second stage, we will filter the traces based on visual observation. Note that for this purpose, we will need to acquire another trace with full read/write addresses for all types of memory traces, unlike Note 2.9. We will refer to such trace as the *full trace*.

We will proceed as follows: first we acquire a full trace and filter it by the mask from the previous method. Then we visualize it and try to find out in which address and temporal range the encryption takes place. Based on our observation, we filter our traces using this range.

Note 2.13. Once we filter the full trace, we get rid of tons of ballast, yielding a trace which is much smaller and easier to understand. Note that the filtered full trace remains correctly aligned to the rest of our traces after filtering.

Trace Visualization. We developed a primitive tool displaying a full memory trace for this purpose. Given n and a trace, this tool splits the trace by addresses into n "compact" segments and displays these parts separately. See Figure 2.3 for an illustrative memory trace of a run of a simple AES implementation. Horizontal axis represents the address range, vertical axis represents the temporal range (from the top).

In this example trace, we can clearly recognize 10 distinct AES rounds even with its 4 subroutines. In general, if possible, we try to estimate the address and temporal range of the first round[4] based on visual observation. In the example, the first round is emphasized by a red rectangle.

[4]Note that we are only interested in the first round, see Equation 2.1 and its context.

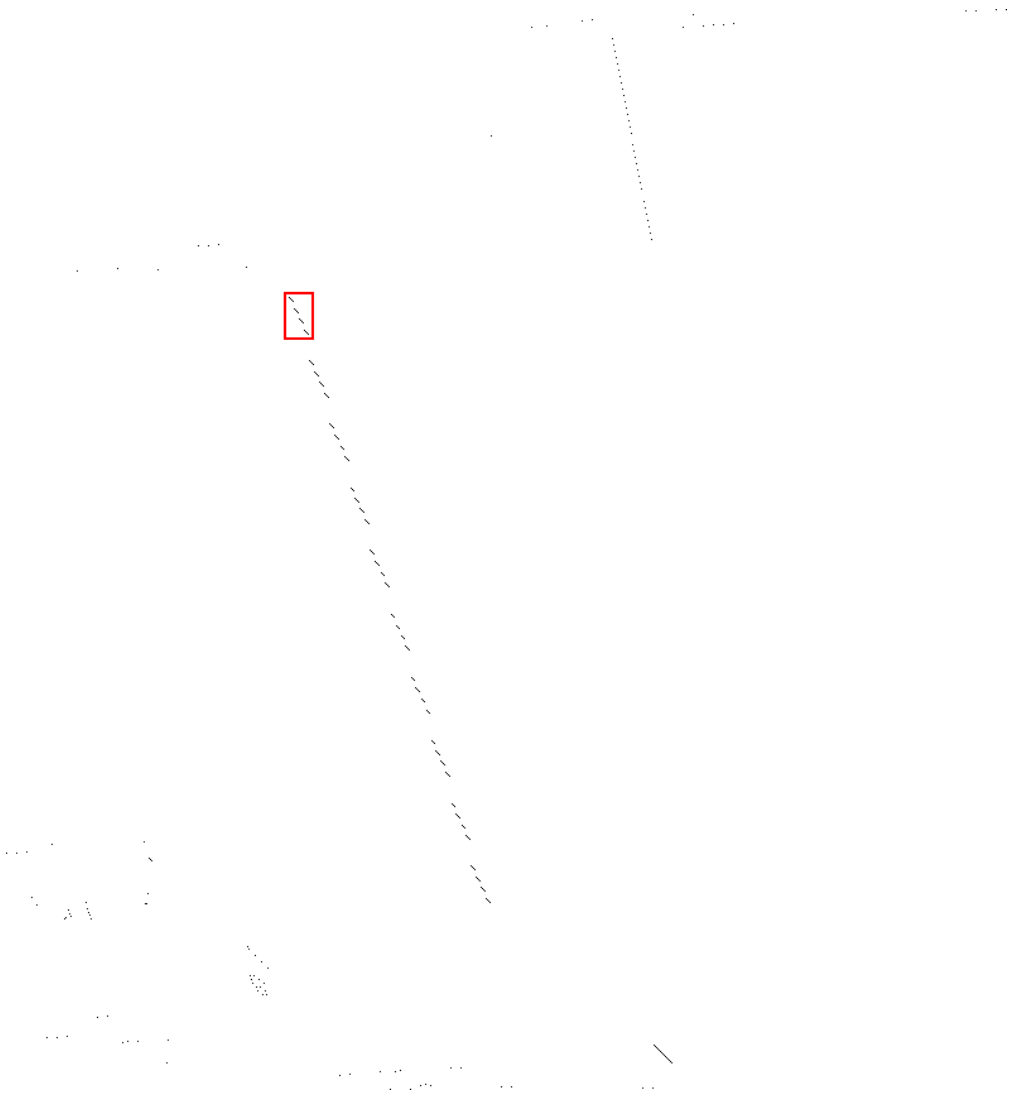

Figure 2.3: Illustrative memory trace of a run of a simple AES implementation with the first round emphasized by a red rectangle. Contents of memory writes were acquired.

Trace Filtering. Given an address and temporal range, we filter our traces so that we only keep their entries that fall into this range. But our traces do not contain full address information, see Note 2.9. The idea could be to use full traces instead.

If we had all full traces, we could filter them easily based on both ranges. But in this case, there may occur a position, for which some traces contain an address falling inside or outside the address range, respectively. Note that this would immediately violate the alignment!

Remark 2.14. There is a better way how to filter our traces by address and temporal range. We simply take our single filtered full trace and create another mask, see Example 2.15. Then we filter our traces with this mask as we did before. This approach ensures that the traces keep aligned.

Example 2.15. Let us say we have deduced from trace visualization that the first round of the AES encryption takes place between rows 0 and 2, and in the address range from `0x7fffe3150000` to `0x7fffe3158000`. Then we get the mask as follows.

Row	Filtered Full Trace	Mask
0	`0x7fffe314f028`	`0`
1	`0x7fffe3150ac8`	`1`
2	`0x7fffe3151dc8`	`1`
3	`0x7fffe3152d18`	`0`
4	`0x7fffe3153948`	`0`

Note that address and temporal filtering brings the most substantial speedup of the whole attack, since its speed depends heavily on the size of the traces. On the other hand, it may happen that we filter our traces too much and loose valuable information, i.e., the attack does not succeed. In such a case, we broaden the address and/or temporal range and rerun again.

Also note that there may occur several 10-part patterns resembling 10 AES rounds. Then we need to keep trying them all until we succeed with our attack.

In the worst case, all our filtering attempts fail or we just cannot see any relevant pattern in the memory trace visualization. Then we omit address and temporal filtering at all and attack for instance only the first byte while, hopefuly, obtaining the position where the key leaks, i.e., where the encryption takes place; here we employ the improved output of our SCA algorithm introduced in Note 2.4. We can look at this position in the visualization and guess some reasonable address and temporal range, use it for filtering and finally attack all 16 bytes with much smaller traces.

2.2.3 Attacking Traces

Once the traces are acquired, possibly filtered and, most importantly, properly aligned – it can be deduced from Note 2.12 and achieved with ideas from Note 2.11 – we can perform the attack. We can use both SCA algorithms presented in this thesis, i.e., Algorithm 2.1 or 2.2, or any other SCA algorithm of our choice.

Remark 2.16. Note that each of our two SCA algorithms was originally designed for a different kind of traces.

In Algorithm 2.1, the traces were supposed to carry information about the Hamming weight of intermediate results, therefore the original attack must be modified to compute

Hamming weights of entries of traces. It might not make much sense to compute the Hamming weight of both intermediate result and trace entries and then their correlation, but this algorithm serves rather as a proof-of-concept. On the other hand, it could possibly bring some surprising results, so we employed it as well.

In Algorithm 2.2, the traces were supposed to be a serialization of eight separate traces, which should carry only bits of information; see Note 2.5. Therefore it is crucial in this attack to consider entries of traces as separate bits, not bytes, i.e., `[1,0,0,0,0,1,0,0, 0,1,1,0,1,1,0,0]` rather than `[84, 6c]`.

As outlined in the previous remark, we will consider Algorithm 2.2 by default since now, unless stated otherwise.

Processing Results

As already stated in Note 2.3, the best key candidate we obtain from a SCA algorithm does not need to be necessarily the correct one, especially when we use it for attacking white-box implementations, where there is no interpretation of what we "measure".

In that note, we suggested to output the full list of key candidates together with their "value" for each key byte. The original purpose of this was to have some rating of key candidates, and then loop through them according to their order and value. In the physical world scenario, this would be likely to recover the key in a relatively short time, since we assume a certain bias in our measurements caused by execution of the cipher.

But it turns out that looping would be useless in the white-box scenario. Indeed, by empirical observation, the true key byte is typically

- either on the top separated by a gap from the second candidate, or
- somewhere in between while the first candidate is separated by a smaller gap,

see the following example with real data. This observation is especially useful for our 8 separate lists; see Note 2.6.

Example 2.17. See Table 2.1 for partial results of a real experiment. According to the original criterion, i.e., picking the candidate with the highest value, we would get `09` in the 5^{th} bit column, but it is not the correct one – in this experiment, the correct candidate does not have the highest value.

On the other hand, if we consider the gap to the second candidate and modify our criterion to take it into account, we can get the correct candidate.

Remark 2.18. According to the observation in the previous example, we can modify our criterion as follows: we choose the candidate that is on the top of any list and has the biggest gap to the second candidate. Note that we can use various methods how to evaluate this gap, e.g., absolute or relative differences of values, and we may obtain different results.

We can further improve this rule by combining (e.g., summing) gaps of the same candidate when it is on the top. On the other hand, note that if the correct candidate is not on the top of any list, improving this rule does not help either.

Rank	4^{th} bit		5^{th} bit		7^{th} bit		8^{th} bit	
	Value	Cand.	Value	Cand.	Value	Cand.	Value	Cand.
0	0.3477	`>15<`	0.3970	`09`	0.3374	`9d`	0.3730	`6f`
1	0.3147	`b5`	0.3884	`d1`	0.3196	`f5`	0.3521	`8b`
2	0.3110	`e7`	0.3699	`65`	0.3183	`37`	0.3451	`7b`
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
23	⋮	⋮	⋮	⋮	0.2895	`fc`	⋮	⋮
24	⋮	⋮	⋮	⋮	0.2892	`>15<`	⋮	⋮
25	⋮	⋮	⋮	⋮	0.2890	`df`	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
41	⋮	⋮	⋮	⋮	⋮	⋮	0.2840	`c1`
42	⋮	⋮	⋮	⋮	⋮	⋮	0.2838	`>15<`
43	⋮	⋮	⋮	⋮	⋮	⋮	0.2825	`b8`
44	⋮	⋮	0.2760	`be`	⋮	⋮	⋮	⋮
45	⋮	⋮	0.2760	`>15<`	⋮	⋮	⋮	⋮
46	⋮	⋮	0.2759	`c4`	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 2.1: Partial results of a real experiment. The true key byte is `15` and is emphasized with angle brackets.

Similarly to the idea of Note 2.3, we can create a list of maximum 8 candidates (one candidate from each of 8 lists) and loop through them to find out whether the key was revealed or not.

Applying the rule from the previous remark to the previous example, we find the maximal gap for the 4^{th} target bit, its absolute value is 0.0330 compared to 0.0086, 0.0178 and 0.0209, its relative value is 9.5% compared to 2.2%, 5.3% and 5.6%, respectively. Hence the best candidate with this rule is `15`, which is the correct one.

Note 2.19. For later use, we recognize two kinds of best candidates according to the gap to the second candidate. If the gap is greater than 10%, it will be referred to as the *strong candidate*, otherwise it will be referred to as the *weak candidate*.

Chapter 3

Results & Improvements

First of all, we describe what tools we needed to perform a practical attack. Then we reproduce previous results in order to have a comparison base for our improvements, which follow. We give results of our contribution and finally conclude with our unifying approach.

3.1 Attack Tools

Since there was no open-source toolkit for the purpose of the DCA described in Section 2.2 at the time of writing[1], we implemented it ourselves. Our toolkit consists of several tools and subtools, see the following list.

1. Acquire traces (see Section 2.2.1):
   (a) acquire a single trace (PIN tools, modified versions of Teuwen's one [53]),
      i. memory reads/writes and their addresses/contents,
   (b) manage trace acquisition (generate & save respective plaintexts along).
2. Filter traces (see Section 2.2.2):
   (a) filter constant entries & save its mask M_C (see Figure 2.2),
   (b) filter by address and temporal range,
      i. acquire another *full* trace T (preferably in the same terminal session),
      ii. filter it with mask M_C, yielding T_C,
      iii. visualize T_C,
      iv. estimate address & temporal range,
      v. create another mask M_R based on the range using T_C,
      vi. filter all traces with M_R (see Remark 2.14).

[1] It was released in March 2016, see [1]. Bos et al. [12] published a link to this toolkit in their March 2016 revision.

3. Attack traces (see Section 2.2.3):
   (a) run attack,
      i. CPA attack (Algorithm 2.1), or
      ii. bitwise DPA attack (Algorithm 2.2),
   (b) process results,
   (c) display results & statistics.

3.2 Results

First of all, we used our toolkit to attack our straightforward C++ implementation of AES, here referred to as `naiveAES`, which was supposed to serve just as a proof-of-concept; see results in Section 3.2.1. Note that Section 3.2.1 does not fully cover many details, these are given later in subsequent sections.

Then we performed the most interesting attack that Bos et al. [12] did – we successfully attacked Klinec's C++ implementation [29] of WBAES by Chow et al. [16], here referred to as `KlinecWBAES`; see results and comments in Section 3.2.2.

Finally in Section 3.2.3, we comment on usage of the attack against a white-box implementation of a cipher derived from WBAES, introduced by Bačinská [9], here referred to as `BacinskaWBAES+`.

Note 3.1. In our results, we only attacked implementations with known source codes and known key. The case of unknown key will be discussed further in Section 4.1.

3.2.1 `naiveAES`

In order to prove the concept of both SCA algorithms presented in this thesis, we first attacked our straightforward AES implementation `naiveAES` with both of them. Both attacks are expected to work, since `naiveAES` stores all intermediate results that are required for each attack. For this reason, we captured contents of all 1-byte memory writes into our memory traces, and used output of the first SBox as our target. In both cases, we treated strong candidates differently (see Note 2.19 for what a strong candidate is).

CPA Attack against `naiveAES`

First we attacked `naiveAES` with the CPA attack (i.e., Algorithm 2.1) using 64, 96 and 128 traces, respectively; see results in Table 3.1 and Note 3.2 for its description.

Traces	Key byte							
	1.		2.		3.		4.	
	5.		6.		7.		8.	
	9.		10.		11.		12.	
	13.		14.		15.		16.	
64	4.2	44	10.2	■	1.8	■	1.6	■
	1.2	6	2.1	10	13.6	1	2.3	6
	10.0	37	10.1	36	19.5	12	0.7	■
	1.8	20	9.1	11	10.7	19	12.7	1
96	10.3	4	5.6	■	6.2	■	12.9	■
	4.5	■	8.8	1	7.2	■	0.4	■
	16.1	11	5.7	3	4.4	2	0.8	1
	0.4	1	2.4	2	0.0	3	8.3	1
128	9.4	■	16.4	■	25.3	■	21.4	■
	4.9	1	13.9	■	24.6	■	11.9	■
	1.8	■	5.8	■	0.9	■	17.5	■
	20.9	■	7.9	■	4.5	1	19.0	■

Table 3.1: CPA attack against `naiveAES` using different number of traces. Percentual gap of the best candidate and rank of the correct candidate is given, for each key byte and each number of traces. The rank ranges from 0, while 0 (i.e., the top position) is replaced with ■ or ■ for strong or weak candidate, respectively, in order to emphasize successful attack.

Note 3.2. In our tables, two values will be given: percentual gap of the best candidate to the second best one together with rank of the correct candidate, for each key byte and each number of traces. Note that the rank ranges from 0 to 255, while 0 is replaced with ■ or ■ for strong or weak candidate, respectively, in order to emphasize successful attack. See the following example for a hint.

Example 3.3. If we looked at the 5^{th} bit back in Table 2.1 in Example 2.17, these two values would be $\frac{0.3970-0.3884}{0.3970} \cdot 100\% \approx 2.2\%$ and 45, respectively.

It appears that we need hundreds of traces in order to attack even an unprotected implementation successfully. Note that in the CPA attack, we only have a single attack target (for each key byte), which is the Hamming weight of the first SBox output (cf. 8 target bits in the bitwise DPA). However, this attack served just as a proof of the concept.

Note 3.4. The term *target of the attack* or simply *target* will be used heavily in the following text, let us describe it a bit more.

Target is a function T that inputs a key byte K together with the respective plaintext byte P, and outputs a byte (for now), or a single bit (later), e.g., $T(K, P) = S(K + P)$

on Lines 6 and 7 of Algorithm 2.2. Target bit is just a specific bit of target's output. According to the value of target bit, traces are split into the sets S_0 and S_1, respectively.

Bitwise DPA Attack against `naiveAES`

Afterwards, we performed the bitwise DPA attack (i.e., Algorithm 2.2) against `naiveAES` using 24 and 32 memory traces, respectively; see results in Table 3.2. The table shows amount of leaking target bits out of 8 of them.

Note 3.5. In case of the correct candidate, the maximum value of the difference of means (which is what the bitwise DPA looks for) is *always* equal to 1.0, since there is always a position in the traces where given intermediate product – here a bit of the first SBox output – appears. Therefore once there appears a non-zero gap for the first time, we certainly get the correct candidate and do not need to care about its strength. Obviously, the attack can be aborted at such moment.

Traces	Key byte															
	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
24	8	7	6	8	7	8	8	8	8	6	8	6	8	8	7	7
32	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8

Table 3.2: Number of target bits that gave single candidate with difference of means equal to 1.0 using the bitwise DPA attack against `naiveAES`.

Using 24 traces has mostly shown to be sufficient, and with 32 traces, the attack was successful for every target bit. Note that these numbers are only illustrative – it could possibly happen that many more traces would keep two candidates on the top, both with the difference of means equal to 1.0.

We presented these results just to give a comparison to the previous CPA attack and to the bitwise DPA attack against a white-box protected implementation.

3.2.2 `KlinecWBAES`

The most interesting results came with attacking Klinec's implementation [29] of WBAES by Chow et al. [16], here denoted as `KlinecWBAES`. Note that we will only consider this particular implementation in the rest of this thesis, except for Section 3.2.3.

First we tried to attack `KlinecWBAES` with the CPA attack, too, but it did not break any key byte at all. Therefore the rest of our results focuses exclusively on the bitwise DPA attack.

Bitwise DPA Attack against `KlinecWBAES`

We used 1024 traces for the attack; see results in Table 3.3. In this table, we give the same values as previously in Table 3.1, see Note 3.2.

Byte	Target bits (percentual gap and rank)															
	1. bit		2. bit		3. bit		4. bit		5. bit		6. bit		7. bit		8. bit	
1.	33.4	■	4.0	55	3.1	90	53.2	■	5.0	149	4.5	207	4.2	224	20.4	■
2.	0.2	248	11.6	218	2.8	239	0.7	244	11.5	247	45.6	■	2.9	251	10.8	247
3.	40.7	■	15.1	212	30.8	■	10.5	25	2.8	230	47.8	■	8.9	99	17.2	■
4.	45.2	■	9.0	252	1.7	226	4.1	247	19.4	■	3.8	255	5.1	241	4.2	252
5.	4.4	247	1.3	104	45.4	■	4.0	225	2.7	229	14.7	■	1.2	225	0.5	249
6.	6.5	252	4.9	255	54.8	■	2.9	241	12.6	242	48.7	■	4.7	4	13.9	255
7.	11.5	47	3.7	233	16.0	■	0.6	228	38.1	■	8.8	■	37.7	■	44.0	■
8.	54.1	■	0.9	253	4.5	253	49.5	■	5.3	255	2.0	251	1.3	1	50.1	■
9.	4.7	224	3.3	196	22.1	231	0.7	249	3.1	253	1.5	238	4.7	■	22.4	253
10.	43.7	■	68.6	■	0.0	255	2.8	245	4.1	255	6.3	■	5.9	234	52.6	■
11.	6.6	245	5.7	■	4.0	250	33.3	■	5.6	190	4.4	255	8.3	236	15.8	■
12.	8.7	254	4.7	255	51.8	■	1.0	255	47.4	■	50.7	■	45.6	■	14.0	■
13.	0.3	241	1.6	■	14.6	254	4.3	190	7.1	160	6.9	193	28.4	■	40.9	■
14.	4.4	235	41.3	■	0.0	254	47.1	■	0.2	255	0.7	2	15.2	■	5.9	255
15.	27.7	■	58.5	■	5.9	246	1.1	195	1.0	255	27.8	■	2.7	246	14.7	155
16.	1.3	252	6.0	255	0.2	254	5.8	■	1.9	251	7.6	245	2.3	235	52.9	■

Table 3.3: Bitwise DPA attack against `KlinecWBAES` using 1024 traces. Percentual gap of the best candidate and rank of the correct candidate is given, for each key byte and each number of traces; see Note 3.2 for a full description.

On average, there are about 31% of successful strong candidates and about 36% of both weak and strong ones.

Compared to the results of Bos et al., we have so far successfully reproduced one half of their attack. The second half is given in Section 3.3.2, because it requires some improvements to be introduced.

Note 3.6. As already noted by Bos et al., we can see remarkably many tail ranks (e.g., between 250 and 255) in results in Table 3.3.

3.2.3 `BacinskaWBAES+`

Bačinská introduced a variant of WBAES in [9], here denoted as `BacinskaWBAES+`. However, this variant is not compliant with the standard AES anymore – SBoxes of `BacinskaWBAES+` are key-dependent, i.e., different from those of the standard AES. Note that the output of the first standard SBox is what the attack is targeting, therefore there is nothing to be successfully attacked.

The attack could be most probably "customized" for `BacinskaWBAES+`. However, there is no practical interest in non-standard ciphers, hence we omitted it.

3.3 Enhancements of the Attack

In this section, we present an improvement by Bos et al. against `KlinecWBAES`, and reproduce their results in order to have a comparison base for further results. Then we generalize the improvement and finally propose our novel framework, which covers all of the previous enhancements of the attack.

3.3.1 Exploiting WBAES Structure

WBAES by Chow et al. has been shown by Klinec [30] to be equivalent to Karroumi's improved version [28]. Note that Karroumi's WBAES internally uses dual AES'es, and therefore we cannot recognize which affine mapping has been used inside SBoxes; see Note 1.62.

Let us first refresh the definition of the original SBox (Equation 1.7), and also note that the multiplication is not performed in F, but in the ring of binary polynomials instead:

$$S(A) = (x^4 + x^3 + x^2 + x + 1) \cdot A' + x^6 + x^5 + x + 1 \pmod{x^8 + 1},$$

where A' stands for the Rijndael inverse of A.

The original SBox is a specific affine mapping of Rijndael inverse of its input, while in Karroumi's approach, this affine mapping is not fixed. Such a new SBox could be written as

$$S_{p,q}(A) = p \cdot A' + q \pmod{x^8 + 1} \tag{3.1}$$

for $p, q \in F$ and p coprime with x^8+1. The simplest affine mapping is obviously achieved for $p = 1$ and $q = 0$, which yields $S_{1,0}(A) = A'$, i.e., only a Rijndael inverse.

Remark 3.7. The reason why p must be coprime with $x^8 + 1$ was already given in the definition of the original SBox in Note 1.47 – it was due to its invertibility. Hence we would like to test whether p is coprime with $x^8 + 1$.

Note that $x^8 + 1 = (x + 1)^8$, therefore if p is *not* coprime with $x^8 + 1$, then 1 must be its root, and vice versa. This can be easily tested, indeed $p(1) = 0$ if the number of its terms is even. It follows that p is coprime with $x^8 + 1$ if and only if the number of terms of p is odd.

3.3.2 Changing Attack Target

Since all of these SBoxes are equivalent inside WBAES, Bos et al. came up with the idea of changing the target of the attack from the original SBox to its simplest variant – Rijndael inverse.

Remark 3.8. Technically we only substitute SBoxes on Lines 6 and 7 of Algorithm 2.2 with their desired variant – here $S_{1,0}$ – hence yielding the following:

6: $S_0 \leftarrow \left\{ Traces[n] \;\middle|\; S_{1,0}(KGuess \oplus PTs[n][i])[j] = 0 \right\}$

7: $S_1 \leftarrow \left\{ Traces[n] \;\middle|\; S_{1,0}(KGuess \oplus PTs[n][i])[j] = 1 \right\}$

Note that we always use precomputed SBox tables as outlined in Remark 2.2.

Results Using Rijndael Inverse

We conducted the same attack as in Section 3.2.2, now with Rijndael inverse as the target (i.e., the simplest SBox variant); see results in Table 3.4. Note that this is the second part of successfully reproduced results of Bos et al.

On average, there are about 27% of successful strong candidates and about 30% of both weak and strong ones.

In both cases (i.e., Tables 3.3 and 3.4), there are, for each key byte, usually either only few target bits that actually leak, or nothing leaks at all (e.g., the 9^{th} byte in Table 3.3 cannot be considered as leaking, since the gap is very small at the 7^{th} bit, and at the 8^{th} bit, the gap is much larger for an incorrect candidate; this happens much more often if less traces are used).

The most substantial practical benefit of attacking different targets is that those key bytes, which did not leak with one target, may possibly leak with another; cf. the 9^{th} byte in both tables.

3.3.3 Considering Another Targets

The benefit of having a higher chance of breaking the key led us to the idea of using yet another invertible affine mappings inside SBoxes (see Equation 3.1) besides the one of the original SBox and identity (i.e., those used by Bos et al.).

There seems to be plenty of them: number of p's is equal to the number of binary polynomials with degree smaller than 8 and coprime with $x^8 + 1$; there are 128 of them (follows from Remark 3.7). Number of q's is simply 256, hence altogether there are

Byte	Target bits (percentual gap and rank)															
	1. bit		2. bit		3. bit		4. bit		5. bit		6. bit		7. bit		8. bit	
1.	3.3	207	45.4	■	9.4	4	7.4	252	2.6	253	11.0	252	43.6	■	35.3	■
2.	5.4	233	1.3	255	0.9	252	49.4	■	4.1	216	8.6	255	10.6	255	47.9	■
3.	0.5	254	9.5	209	20.6	■	45.6	■	7.0	254	0.4	225	8.9	247	2.8	189
4.	2.1	37	12.7	■	10.7	251	2.0	■	7.2	■	4.9	252	0.7	231	9.0	242
5.	2.1	244	17.9	■	5.7	250	0.4	231	2.5	134	2.0	79	5.8	214	3.6	223
6.	53.2	■	1.3	253	8.9	255	9.0	254	38.2	■	37.8	■	43.6	■	7.3	2
7.	24.5	■	0.9	248	6.4	187	5.8	255	13.6	209	36.2	■	0.9	184	2.7	227
8.	48.9	■	7.4	255	4.2	255	6.5	242	8.1	234	1.9	253	47.2	■	2.8	255
9.	0.3	227	0.6	156	6.0	237	2.5	243	14.4	229	6.2	232	51.3	■	15.1	■
10.	8.9	■	3.0	158	10.1	1	40.5	■	4.2	253	12.2	■	54.3	■	50.1	■
11.	4.7	248	51.7	■	6.7	241	0.4	254	4.4	251	5.2	45	10.1	255	4.7	1
12.	43.2	■	2.8	251	7.2	254	2.5	255	6.9	236	3.2	255	49.2	■	6.2	254
13.	7.3	205	6.2	4	9.2	191	6.6	30	63.7	■	14.7	■	7.5	240	0.8	255
14.	61.4	■	4.9	231	1.7	246	54.4	■	1.9	248	9.7	253	15.8	■	46.2	■
15.	0.7	221	2.0	250	4.7	1	2.0	■	8.0	223	31.3	■	1.4	1	21.0	225
16.	4.0	255	57.7	■	6.5	229	4.3	254	47.1	■	9.5	255	8.2	254	1.7	253

Table 3.4: Bitwise DPA attack against `KlinecWBAES` using 1024 traces, Rijndael inverse taken as the target. Percentual gap of the best candidate and rank of the correct candidate is given, for each key byte and each number of traces; see Note 3.2 for a full description.

$128 \cdot 256 = 32\,768$ invertible affine mappings, which could be used inside SBoxes to create another targets.

Remark 3.9. We realized that there are certain classes of mappings which give very the same results.

Effect of q's. First, let us see what happens if we change one bit of q considered as a byte. It obviously only changes the output of our SBox at the same bit, finally yielding a swap of S_0 and S_1 in the attack algorithm at this target bit.

Note that this swap has no effect on results, since we are only interested in absolute difference of means of values inside S_0 and S_1. Hence we will set $q = 0$, and our SBoxes will be just linear mappings of Rijndael inverse.

Effect of p's. Second, let us study the effect of different p's. Note that x is coprime with $x^8 + 1$ and p is supposed to be coprime, too, therefore if we multiply p by $x \pmod{x^8 + 1}$, the result is still coprime with $x^8 + 1$.

Now let us see what multiplying by $x \pmod{x^8+1}$ actually does. If the result does not need to be reduced, it simply shifts coefficients of p by one, e.g., $(x^4+x+1)\cdot x = x^5 + x^2 + x$. If it reaches x^8, e.g., $(x^7 + x^4 + x^3) \cdot x \pmod{x^8 + 1} = x^5 + x^4 + 1$, we get 1 at the end, so the shift is actually a cyclical shift by one.

Note that polynomials coprime with $x^8 + 1$ have odd number of terms (see Remark 3.7), hence repeating multiplication by x yields 8 distinct polynomials, which we put into single equivalence class. Since there are 128 of such coprime polynomials, we get $128/8 = 16$ of such classes; see Table 3.5 for representants of each class. Note that the effect is the same on bytes representing polynomials.

Since we assumed $q = 0$, the actual output of two SBoxes using p's from a single class is thus also only a cyclical shift of each other. It follows that the results are also only cyclically shifted across the target bits. Hence the information, which target bit leaks, is totally irrelevant, too.

01	07	0b	0d	13	15	19	1f
25	2f	37	3b	3d	57	5b	7f

Table 3.5: Representants of classes of p's in hexadecimal form. Note that p of the original SBox is `1f`.

It follows that we can narrow down our attention to 16 targets only. Indeed, our targets will be of the form $T(K, P) = S_{p,0}(K + P)$ for all representants p from Table 3.5. Note that Bos et al. only used the original SBox (i.e., $p =$ `1f`) and Rijndael inverse (i.e., $p =$ `01`).

Results Using the 16 Targets

We attacked `KlinecWBAES` using the 16 attack targets and 1024 traces; see results of the 1st byte in Table 3.7. Note that the row of target `01` is indeed equal to the row of the 1st byte in Table 3.4, and the row of the target `1f` is indeed equal to the row of the 1st byte in Table 3.3. Also note that the target `25` does not unravel the 1st key byte even with 1024 traces – here we can clearly see the benefit of having many targets.

3.3.4 Non-Invertible Linear Mappings

Other results emerged with the idea of trying to use also non-invertible linear mappings inside the attack targets. Note that the mappings were originally required to be invertible, otherwise the resulting cipher would not be invertible. But there is no such limitation on attack target – the attack is actually only required to work, no matter why.

As before, we only kept one representant of each class, where elements of such class are multiples of $x^i \pmod{x^8+1}$ of each other, for some i; see these representants in Table 3.6.

00	03	05	09	0f	11	17	1b	1d	27
2b	2d	33	35	3f	55	5f	6f	77	ff

Table 3.6: Representants of classes of non-invertible p's in hexadecimal form.

We can immediately discard `00`, because it only gives 0 (everything would fall into S_0 in the attack algorithm). Also note that some classes include less than 8 polynomials, e.g., `11` $= x^4+1$. This is because $(x^4+1)\cdot x^4 \pmod{x^8+1} = x^4+1$. For this reason, some target bits are not shown in results since their values would repeat.

We attacked `KlinecWBAES` using the new non-invertible targets and 1024 traces; see results of the 1st byte in Table 3.8.

3.3.5 Unifying Approach in $\mathrm{GF}(2)^8$

Let us have a closer look at our previously considered targets, in particular at matrix representation of their internal linear mappings – multiplication by $p \pmod{x^8+1}$ – in $\mathrm{GF}(2)^8$ (quick reminder: $T(K,P) = S_{p,0}(K+P) = p\cdot(K+P)' \pmod{x^8+1}$). For this purpose, let us denote the vector space $\mathrm{GF}(2)^8$ over $\mathrm{GF}(2)$ by $\mathcal{B}$.

Note that equivalents of multiplication by $p \pmod{x^8+1}$ form a specific subset of all linear mappings on $\mathcal{B}$ – simply since there are only 256 of p's, but there are obviously many more linear mappings on $\mathcal{B}$. Let us see how multiplication by $p \pmod{x^8+1}$ can be expressed in $\mathcal{B}$ – its matrix has cyclically shifted rows as outlined in the following example[2].

[2]Note the analogy of `MixColumns` – it was originally defined as modular polynomial multiplication in Equation 1.8 and further represented by matrix multiplication in Equation 1.9, where the matrix had

Target	Target bits (percentual gap and rank)															
	1st byte															
01	3.3	207	45.4	■	9.4	4	7.4	252	2.6	253	11.0	252	43.6	■	35.3	■
07	10.8	254	3.3	■	5.2	246	56.6	■	1.1	206	40.2	■	6.3	208	1.7	239
0b	38.0	■	2.6	254	14.5	254	1.5	250	5.9	216	1.8	254	37.5	■	57.3	■
0d	54.6	■	0.5	250	10.2	■	19.1	■	2.9	140	0.2	222	0.1	234	8.4	202
13	27.4	■	7.0	217	49.5	■	5.2	254	0.9	185	11.3	■	18.3	230	13.8	201
15	3.0	238	24.1	■	19.6	252	3.9	236	0.9	191	3.9	251	4.7	1	31.8	■
19	0.6	242	8.1	1	2.3	201	7.2	240	40.2	■	12.4	188	7.9	251	1.9	242
1f	33.4	■	4.0	55	3.1	90	53.2	■	5.0	149	4.5	207	4.2	224	20.4	■
25	10.9	253	0.0	254	14.2	212	2.0	233	4.2	145	7.3	218	3.4	1	9.0	251
2f	51.4	■	7.1	222	0.4	249	10.1	233	2.6	253	0.1	254	3.3	213	21.4	253
37	8.8	247	7.8	■	2.2	236	8.9	252	36.6	■	7.4	253	4.6	254	1.0	244
3b	7.6	228	40.5	■	3.2	146	3.8	235	2.3	228	2.0	147	40.0	■	41.6	■
3d	18.7	253	11.5	225	14.6	■	19.5	■	1.3	232	18.7	221	25.9	200	5.2	184
57	3.6	243	1.0	244	1.5	251	4.3	209	3.4	2	48.9	■	0.5	254	10.6	244
5b	45.5	■	7.6	225	32.3	■	1.9	250	4.4	183	4.4	217	3.2	250	2.0	229
7f	2.7	220	2.6	246	4.8	241	1.3	196	4.3	234	49.3	■	20.6	215	2.0	147

Table 3.7: Bitwise DPA attack against `KlinecWBAES` using 1024 traces and 16 invertible targets.

Target	Target bits (percentual gap and rank)															
	1st byte															
03	6.3	94	6.4	253	38.9	■	2.1	180	1.2	8	1.6	237	2.6	189	2.5	232
05	4.9	148	0.2	255	6.1	212	43.6	■	4.1	6	24.7	■	5.0	202	4.9	215
09	9.2	255	4.0	172	12.2	207	55.8	■	44.3	■	10.1	232	2.0	190	7.6	84
0f	3.2	253	34.1	■	4.7	224	3.1	238	0.9	207	7.2	212	0.9	232	0.6	238
11	5.5	17	1.9	240	5.7	206	2.2	186	—		—		—		—	
17	14.8	■	33.7	■	2.9	233	1.4	252	2.0	236	2.0	226	3.8	■	6.2	246
1b	3.6	63	41.0	■	26.4	■	26.0	251	14.1	■	1.0	255	58.1	■	6.6	155
1d	28.1	■	52.9	■	38.0	■	34.9	■	60.8	■	3.2	253	5.0	157	51.1	■
27	0.8	1	4.7	247	11.1	247	1.2	217	9.1	229	10.8	242	56.4	■	1.4	104
2b	5.9	245	4.9	■	4.7	250	48.4	■	3.1	222	0.1	246	56.2	■	8.4	247
2d	0.5	221	0.1	196	49.3	■	1.3	239	18.7	■	4.8	196	0.7	245	7.8	■
33	0.2	221	11.7	■	10.9	■	5.0	215	—		—		—		—	
35	4.8	244	8.4	228	47.9	■	13.7	211	2.6	250	2.0	115	24.5	■	18.5	■
3f	5.2	249	3.7	109	4.9	■	1.9	76	2.3	222	3.1	254	4.3	189	6.3	250
55	9.8	220	2.0	239	—		—		—		—		—		—	
5f	6.1	225	5.7	245	39.6	■	2.4	251	0.8	252	4.4	3	0.6	231	4.7	237
6f	3.4	86	34.3	■	1.3	243	1.7	234	24.1	■	17.5	■	0.2	218	0.4	244
77	43.7	■	2.2	219	0.1	194	7.5	130	—		—		—		—	
ff	0.0	223	—		—		—		—		—		—		—	

Table 3.8: Bitwise DPA attack against `KlinecWBAES` using 1024 traces and non-invertible targets.

Example 3.10 (Multiplication by `3d` $(\bmod x^8+1)$)**.**
`3d` can be rewritten as `3d` $= x^5+x^4+x^3+x^2+1 \sim 00111101 \in \mathcal{B}$, let further $A = a_7x^7+\ldots+a_1x+a_0$. Then we have

$$\begin{aligned}
\texttt{3d}\cdot A \mod x^8+1 &= (x^5+x^4+x^3+x^2+1)\cdot(a_7x^7+\ldots+a_1x+a_0) \mod x^8+1 = \\
&= (a_7+a_5+a_4+a_3+a_2)\cdot x^7 + \\
&\quad \vdots \\
&\quad + (a_7+a_6+a_5+a_4+a_1)\cdot x + \\
&\quad + (a_6+a_5+a_4+a_3+a_0)\cdot 1 \sim \\
&\sim \begin{pmatrix} \boxed{10111100} \\ 01011110 \\ 00101111 \\ 10010111 \\ 11001011 \\ 11100101 \\ 11110010 \\ 01111001 \end{pmatrix} \cdot \begin{pmatrix} a_7 \\ \vdots \\ a_1 \\ a_0 \end{pmatrix},
\end{aligned}$$

where `3d` can be found reversed in the first row of the corresponding matrix.

One may wonder whether we could get yet another set of targets, but actually we do not. Indeed, let us take a matrix $\mathbb{A}$ of an arbitrary linear mapping and observe the j-th bit of its output (suppose that the j-th row of $\mathbb{A}$ is non-zero). Then this output bit is a scalar product of the j-th row of $\mathbb{A}$ and the input.

Since the attack performs bitwise, we only care about the j-th row of $\mathbb{A}$. But note that there exists very the same row in a matrix representing some of our previously considered linear mappings – let us say on the i-th row of a matrix denoted by $\mathbb{B}$. Indeed, we used all possible non-zero bytes and their equivalence classes were inner cyclical shifts. It follows that if we attack the i-th target bit using $\mathbb{B}$, we get very the same results as with the j-th target bit using $\mathbb{A}$, hence we do not get any new targets.

As outlined before, we are only interested in non-zero vectors; there are 255 of them. Note that there are indeed 255 entries altogether in Tables 3.7 and 3.8.

The output of this kind of target is a scalar product of two vectors, hence not a byte anymore, but a single bit (as mentioned in Note 3.4). Let us express such target according to the notation of Note 3.4 as

$$T_B(K,P) = [B]^T \cdot [(K+P)'], \tag{3.2}$$

where $B \in \mathcal{B}$ stands for any non-zero vector, $(\cdot)'$ for the Rijndael inverse, $[\cdot]$ for column vector representation and $[\cdot]^T$ for vector transposition, hence both vectors are scalar-multiplied.

---

cyclically shifted rows.

Even though this observation does not bring any new targets, its benefit is a unifying and simplifying approach. Hence we do not give any results, since these are already given in Tables 3.7 and 3.8 as explained before.

We only wondered whether we should split our remarks in the following Section 4.1.1 by the origin of the corresponding linear mapping (i.e., either invertible, or non-invertible) and finally decided to do so, because there could possibly emerge some surprising results.

Implementation Note

Now, as we have 255 targets outputting only a single bit, we have to modify our attack, i.e., Algorithm 2.2. We skip the `for` cycle on Line 5 and also modify the following Lines 6 and 7 into the following form:

6: $S_0 \leftarrow \left\{ Traces[n] \;\middle|\; T_B(KGuess, PTs[n][i]) = 0 \right\}$

7: $S_1 \leftarrow \left\{ Traces[n] \;\middle|\; T_B(KGuess, PTs[n][i]) = 1 \right\}$

where T_B is the mapping introduced in Equation 3.2. For this purpose, we precompute 255 lookup tables of T_B's, for all B's, of course.

Chapter 4

Analysis of the Attack against WBAES

In this chapter, we present some remarks regarding the attack introduced in Section 3.3.5, i.e., the attack against Klinec's implementation [29] of WBAES by Chow et al. [16], denoted by `KlinecWBAES`. Based on our observations, we suggest a blind attack (i.e., an attack without knowing the actual key) and mention some consequences in white-box design. Finally, we outline, justify and confirm our explanation of the attack.

4.1 Blind Attack

In a real-world scenario, the key is typically unknown. Therefore we do not have any information whether our best candidate is correct or not. We need some rule to recognize a fairly good candidate, which we can then test for correctness by running AES encryption ourselves.

The most straightforward rule seems to be thresholding the gap of the best candidate, but it turns out that it is not that easy, see the following remark.

Remark 4.1. Especially note the attack against the 1^{st} byte using `3d` as a target and its last but one entry in Table 3.7 – here an incorrect candidate has a gap of almost 26%. It is actually the largest gap of an incorrect candidate among all targets and all bytes using this particular instance of WBAES tables. (The largest gap of an incorrect candidate ever seen was almost 35%!)

In our results, the target bits keep their original order (but reverse), therefore we can deduce the vector B generating this particular target in terms of T_B as introduced in Equation 3.2. Here it is the second row of matrix of multiplication by `3d` $(\text{mod } x^8 + 1)$, see Example 3.10. Hence the vector is $B = 01011110$.

In order to perform a blind attack, we studied our results deeper – we were particularly interested in success rates of individual targets. The following section gives some of our remarks, next we suggest the blind attack itself in Section 4.1.2

4.1.1 Remarks

We obviously cannot make conclusions based on a single `KlinecWBAES` table instantiation only. In order to avoid specific behavior of a single instance, we created other 7 instances. Then we acquired 1024 traces and ran the attack against each of 16 key bytes using all 255 targets, for each instance. Altogether we ran $16 \cdot 255 \cdot 8 = 32\,640$ attacks, which, together with trace acquisition and filtering, took us several hours on a regular hardware.

We processed the results and displayed several statistics, here we present some of them. Note that we only considered strong candidates as introduced in Note 2.19 (candidates with a gap greater than 10%).

The good news is that the overall success rate was 25% (cf. 27% and 31% for our previous single-instance attack using the original SBox and Rijndael inverse, respectively), which gives us a hope that the new targets are successful, too. But our goal is to estimate rather individual success rate of each of our 255 targets – here the good news is that each target was successful at least once out of $16 \cdot 8 = 128$ chances.

Let us give the remarks now – namely we present success rates of our targets, behavior of incorrect candidates, what happens if we use less than 1024 traces, and finally we show a very peculiar result regarding leaking position within our traces.

Success Rate of Individual Targets

Note that we have 255 targets and only 8 independent[1] instances of tables, which is quite a small data set for statistical purposes. For this reason, we rather group targets by different criteria and observe if they appear to be uniformly distributed. Note that we could possibly only refute uniformity this way, but it can still serve as a reasonable justification.

Group by corresponding p. Here we make use of our former approach – group the targets by corresponding p, and also study whether there is some influence of target invertibility; see the transfer in Section 3.3.5. We put the results in a histogram, see Figure 4.1. Here we give average percentual success rate together with its standard deviation (among 8 measurements), for each group of targets. Invertible p's are shown in green, non-invertible ones are orange.

Note that we did not give y-axis scale, since the only purpose is to distinguish uniform distribution, which appears to achieved be here.

Group by fixed 4 bits of B. Another way of grouping we performed was grouping targets by fixing 4 out of 8 bits of their corresponding vector B. We fixed the first and the last 4 bits and grouped them; see results in Figure 4.2, where these ways of grouping are given in green and orange, respectively. Note that group index can be computed as binary AND of mask `0xf0` and `0x0f` with vector B, respectively.

[1] At the moment, we do not know which intermediate product leaks, hence cannot claim which values within a single instance are independent. Clearly, more of them could be possibly independent.

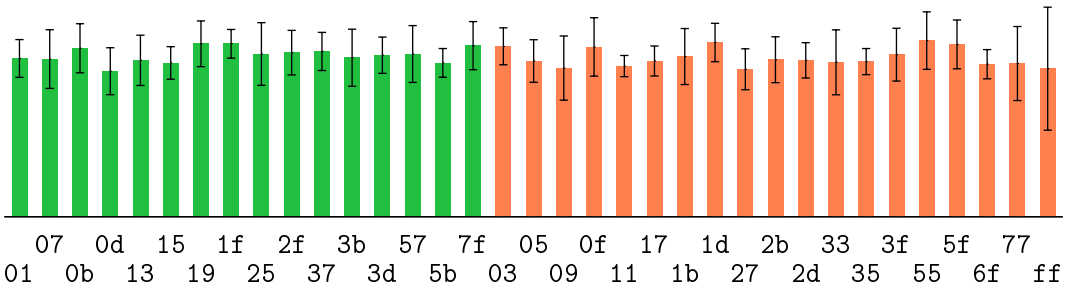

Figure 4.1: Average success rate and its standard deviation for targets grouped by corresponding p. Green represents invertible p's, orange non-invertible ones.

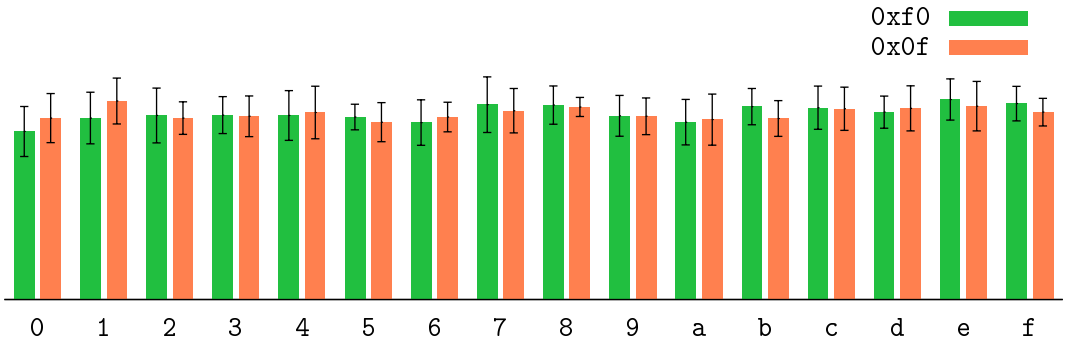

Figure 4.2: Average success rate and its standard deviation for targets grouped by fixed 4 bits of corresponding vector B. Green represents fixed first 4 bits, orange represents fixed last 4 bits.

Remark 4.2. According to our previous observations, there does not seem to be a preferred target or a group of targets, therefore we will assume that the success rate is uniform across all targets.

Incorrect Candidates

There is an inconvenience, which emerges once we do not know the key – we do not recognize the correct candidate anymore, see Remark 4.1. Therefore we will first observe the behavior of the incorrect ones in our self-controlled experiment where we know the key.

Gaps of incorrect candidates. Previously we defined a strong candidate as a candidate with a gap greater than 10%. Most strong candidates were correct, too, but there were a couple of exceptions, which we will refer to as *false positives*. In our overall

results, 25% of candidates were strong and correct, on the other hand, there were 11% of false positives. On average, false positives had a gap of 14% (cf. 40% for correct ones) and the highest gap ever seen was 35% (cf. 76% for correct ones).

A criterion based on thresholding might be on the one hand too strict, on the other hand, it might fail. Instead of looking for some well-balanced threshold, let us have a look at a much better observation about false positives, which will help us to distinguish them easily from the correct ones.

Number of repetitions of false positives. The good news is that the same false positive does not appear to repeat across targets very often. We observed the number of most repeating false positives across our 255 targets and got the following results: on average, the number was 1.75, the global maximum was 3.

Therefore having a candidate that repeats several times, we most likely have the correct candidate.

Using Less Traces

So far, we always used 1024 traces in our attacks, let us have a look at results of the attack with much less traces. Note that Bos et al. [12] used 2000 and 500 traces for the attack targeting the original SBox and Rijndael inverse, respectively.

Namely, we used 128, 256, 384 and 512 traces. For each number of traces, we attacked all of our 8 instances of `KlinecWBAES`, and observed ratios of strong candidates (both correct and incorrect) and their average gap; see results in Table 4.1. Note that number of repeating false positives remains very low – up to three.

Traces	128	256	384	512	1024
Correct candidates	9.8%	17%	20%	21%	25%
Average gap of correct candidates	24%	32%	35%	37%	40%
False positives	7.7%	11%	12%	12%	11%
Average gap of false positives	13%	14%	14%	14%	14%
Reduced cost[2]	5 400	4 700	5 500	6 400	10 500

Table 4.1: Ratios of correct and incorrect candidates and their average gaps, for different numbers of traces.

Within-Trace Position of Leaks

Note that our trace is a bitwise serialization of least significant bytes of memory addresses, hence we can identify which bit within that byte was leaking – simply by taking the leaking position within trace modulo 8.

[2]Reduced cost is to be introduced later.

Note 4.3. This moduled position will be referred to as *leaking bit* (note the difference from target bit).

We noticed soon that leaking bits are not very well balanced as one would expect to, therefore we put this data into a histogram; see Figure 4.3. The histogram shows, for each leaking bit, the average number of strong correct candidates together with its standard deviation, which is surprisingly very low. Note that the histogram does not provide y-axis scale for similar reasons as for previous histograms.

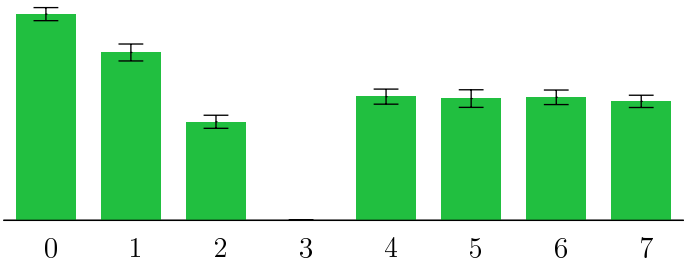

Figure 4.3: Average number of leaks and its standard deviation at each leaking bit.

The 0^{th} and 1^{st} bits leaked slightly more, the 2^{nd} bit slightly less than average, and the 3^{rd} bit actually leaked only twice, while the overall average of the remaining bits was almost 160! On the other hand, all of the remaining bits leaked fairly similarly. We do not have any explanation for this behavior, we only present it as our observation.

4.1.2 Blind Attack Suggestion

Note 4.4. This section only applies previous observations on a heuristic basis, there is no guarantee that our approach is the best.

In general, we suggest to use rather less traces, and repeat the attack with several targets until the sum of strong gaps of any candidate exceeds given bound. Let us choose reasonable values for both the number of traces and the bound.

Number of traces. In order to deduce a reasonable number of traces from results in Table 4.1, let us introduce *reduced cost of gap*, defined as

$$C(n, s, g) = \frac{n}{s \cdot g},$$

where n stands for the number of traces, s for average success rate and g for average gap of a strong candidate.

Note that this quantity corresponds with average time of the attack. Indeed, the more traces, the longer time; the better success rate or the bigger gap, the shorter time. Hence we can use this quantity to estimate the expected time of the attack using different number of traces.

According to results in Table 4.1, the best value of the reduced cost of gap was achieved for 256 traces, therefore we suggest to use 256 traces.

Bound. According to our previous observation, false positives do not appear to repeat very much – there were only a couple of repetitions for the most successful ones[3]. Therefore we suggest to use 75% as a cumulative bound, since it requires some 6 hits for a fairly successful false positive, which appears to be hardly achievable.

Remark 4.5. The attack can be further improved – we can make the bound adaptive: we begin with a lower bound, obtain some (full) key candidate and verify it. Until the candidate is correct, we keep increasing the bound and merge our previous and new results. However, we did not implement this enhanced variant.

4.2 Use in White-Box Cryptography Design

The benefit of this attack is obvious – it introduces a principially different approach to break white-box cryptography implementations. This attack may help to address weaknesses of any future white-box cryptography implementation and possibly increase its resistance to this kind of attack.

So far, we were only interested in results of the attack. In this section, we will study practical consequences of the attack itself. First, we describe our initial effort trying to identify which intermediate product causes the leakage, then we outline some countermeasures against this attack. Both might be useful in white-box cryptography design.

4.2.1 Point of Leakage

Note that the attack allows us to reveal concrete addresses that contributed to the key recovery. Our original intention was to find the corresponding piece of source code, and possibly deduce the intermediate product that is later transformed into leaking memory address, – it would most likely be used as an array index.

We tried two ways of using a debugger how to find that position in the source code, but neither of them gave any results. Authors of the DCA toolkit [1] were successful at this point, see their comprehensive and very technical report on a wiki page of their Deadpool repository[4].

One could possibly try yet another approach – modify the source code to output directly all relevant array indexes. Note that it might be challenging to address them really all.

However, we did not finally make use of this information (i.e., the point of leakage in source code) – we identified the leaking intermediate product rather by a mathematical reasoning. This will be given in Section 4.3.

[3] This holds also when using less traces.

[4] Direct link:
`github.com/SideChannelMarvels/Deadpool/wiki/Tutorial-%234%3A-DCA-against-Karroumi-2010-challenge`.
Accessed: 2016-04-24.

4.2.2 Countermeasures against DCA

Since the DCA attack is based on an algorithm that was originally developed for physical measurements of certain hardware emissions, we can look for some inspiration back in hardware model. There are several countermeasures, see the following list for some of them (basic ones are introduced in [14, 24]).

Masking with random values. Cryptographic hardware can run some unpretendable source of random data and thus mess up values in traces (i.e., adding a strong noise). However, there is no equivalent of noise in the white-box context, since everything is is fully controlled by the attacker.

Reordering instructions. Note that our algorithms required aligned traces, therefore it would be fatal if the leaking position were at several different places. However, this could be possibly overcome in the white-box context – we could achieve this by reverse engineering or, much easier, by aligning the traces with respect to instruction address trace (as outlined by Bos et al.).

Adding random delays. Note that random delays have very similar properties as the previous countermeasures – it can be controlled by the attacker and it only causes a trace misalignment.

In general, we cannot rely on any source of *dynamic* random data (generated during program execution). All we need to rely on is *static* random data (generated during instantiation). On the other hand, dynamic randomness can be used in a white-box implementation – just as another level of "obfuscation".

Bos et al. further propose to use some ideas from *threshold implementations* [43] or use of external encodings – here they emphasize that more research is needed.

4.3 Explanation Attempt

In this section, we identify the leaking intermediate product – we present both reasoning and experimental confirmation.

First of all, let us refresh the information flow through WBAES tables (originally in Equation 1.11), especially its first table,

$$\ldots \to \underbrace{\texttt{Enc} \to \texttt{IMB}^{-1} \xrightarrow{\text{plain}} \texttt{TBox} \xrightarrow{\text{plain}} \texttt{MB} \circ \texttt{MC} \to \texttt{Enc}^{-1}}_{\text{in table}} \to \ldots.$$

Note that the last plain AES intermediate product is actually the 8-bit output of the first SBox, i.e., the very original target. Let us see what happens next: a composed linear mapping $\texttt{MB} \circ \texttt{MC}$ is applied resulting in a 32-bit output (using an idea outlined in Section 1.3.3), then each 4-bit nibble is passed through a 4-bit random bijection $\texttt{Enc}^{-1}$ (further simply $\texttt{Enc}$).

MB ∘ MC. We can view each bit t of the 32-bit output of MB∘MC as a binary scalar product of a row $[R]^T$ of matrix representing MB ∘ MC with its 8-bit input $[S]$, which is actually the output of the first SBox (note that MB is a random linear mapping with certain restrictions, i.e., $[R]^T$ is a non-zero random-like 8-bit vector). We get

$$t = [R]^T \cdot [S] = [R]^T \cdot \Big(\{\texttt{1f}\} \cdot [(K+P)'] + [\texttt{63}]\Big),$$

where $\{\cdot\}$ stands for binary matrix corresponding to multiplication modulo $x^8 + 1$, $(\cdot)'$ stands for the Rijndael inverse, and K and P stand for a key byte and corresponding plaintext byte, respectively, i.e., there is the output of the first SBox in the big parentheses; cf. Equation 1.7. This equation can be easily turned into the form of previously used target (Equation 3.2), indeed

$$\begin{aligned}[R]^T \cdot \Big(\{\texttt{1f}\} \cdot [(K+P)'] + [\texttt{63}]\Big) &= ([R]^T \cdot \{\texttt{1f}\}) \cdot [(K+P)'] + ([R]^T \cdot [\texttt{63}]) = \\ &= [\bar{R}]^T \cdot [(K+P)'] + \bar{r}, \end{aligned} \tag{4.1}$$

where $\bar{R}$ is just a different random-like vector, which plays the role of B in Equation 3.2, and $\bar{r}$ is a constant bit. Note that additive constant does not need to be considered inside target (see Remark 3.9). It follows that each of these 32 bits perfectly matches some of our 255 targets introduced in Section 3.3.5.

Note that, even though MB ∘ MC introduces diffusion, there is no diffusion so far, because those 32-bit intermediate results have not been combined together yet. Therefore these intermediate results only depend on a single byte of both key and plaintext!

If we could directly observe these intermediate results, we would get the difference of means equal to 1.0 (cf. attack against unprotected implementation in Section 3.2.1, see Note 3.5).

Remark 4.6. It follows that the protection against our 255 targets is fully and solely accomplished by Enc.

Enc. Note that Enc is a (only) 4-bit random bijection. Note that, unlike confusion elements in regular ciphers (e.g., SBox in AES), there is no nonlinearity check[5] in WBAES by Chow et al.

The choice of Enc actually only relies on the fact that the ratio of affine mappings among all random bijections is extremely low (less than 0.000 002% for 4-bit bijections). This justification is very poor, since it does not even address the case where one output bit of Enc is an affine mapping of the input!

There are actually quite many bijections that are affine in a single output bit. Indeed, there are $2 \cdot 4 \cdot (2^4 - 1) \cdot 8! \cdot 8! \approx 2 \cdot 10^{11}$ of them among $16! \approx 2 \cdot 10^{13}$

[5] Linearity in ciphers is studied by *linear cryptanalysis*, see, e.g., [37].

bijections on 4-bits altogether, which already makes some 1% ratio! Also note that in such case, the maximum difference of means would be still 1.0, since this affine mapping could be composed with the previous affine mappings, yielding an affine mapping again.

But we do not necessarily even need a fully affine mapping at some bit – it would be sufficient if the mapping were affine for most inputs. Clearly, there are much more such mappings. For this reason, this intermediate result of WBAES seems to be most likely that causing the leakage.

In order to support our hypothesis, we modified the source code of `KlinecWBAES` to output these intermediate products (i.e., outputs of the first Type II tables, see Equation 1.11) and ran the attack using these values instead of memory traces. The attack indeed succeeded, which fully supports our explanation.

Note 4.7. Our explanation further appears to support our conjecture on uniform distribution of success rates across targets as introduced in Remark 4.2. Indeed, there is no evidence of preference of any target, which is the role of $\bar{R}$ in Equation 4.1, further possibly composed with partially linear `Enc`. However, we do not prove it rigorously.

On the other hand, we noticed an interesting difference from our previous results: leaking bits, as introduced in Note 4.3 and unlike previous observations (see Figure 4.3), appear to be uniform in this case! We still do not have any explanation for this behavior.

Chapter 5

Future Work

5.1 Another Interesting AES White-Box Implementation

Bos et al. [12] mention another interesting AES white-box implementation in their paper. This particular implementation by Eloi Vanderbéken was originally published only as a Windows binary executable challenge at NoSuchCon 2013 conference[1], where nobody was able to recover the key. Note that this implementation uses an externally-encoded variant of AES (as outlined in Section 1.4.4, in particular in Equation 1.12), hence it is *not compliant with AES anymore*. It follows that any SCA-like attack is impossible, since the internal AES handles encoded inputs, where the encoding is unknown.

Soon after the conference, the author released source codes[2], which allow one to avoid reverse engineering of the binary. Moreover, it allows one to modify the code to run natively on Linux. Philippe Teuwen, a co-author of [12], studied this implementation deeper, and finally was able to recover the key [54].

In his write-up, Teuwen also describes the structure, which appears to avoid usage of 4-bit random bijections (as used in WBAES by Chow et al. [16]) and uses 8-bit ones instead. Note that even without nonlinearity check, it is far less likely that any output bit of a random 8-bit bijection would provide a reasonable level of linearity compared to a random 4-bit bijection, as outlined in Section 4.3.

However, we did not either modify the source code to make it AES-compliant, nor did run the attack, hence it remains as a possible topic for further research.

5.2 Miscellaneous Remarks

During writing of this thesis, there emerged a couple of unresolved issues. Here we give a short list of those, which we think would deserve further attention, in the order of their significance from our perspective.

[1] `nosuchcon.org/2013`. Accessed: 2016-04-15.
[2] `pastebin.com/MvXpGZts`. Accessed: 2016-04-15.

(Semi-)automatic leakage locator. In Section 4.2.1, we mentioned a successful attempt to find the leaking position in source code. A (semi-)automatic tool would be definitely appreciated in white-box cryptography design.

Adaptive bound. In Remark 4.5, we outlined a possible enhancement of our attack with adaptive bound, which would certainly accelerate the attack.

(Non)uniformity in leaking bits. Leaking bits, as introduced in Note 4.3, indicate quite a weird behavior in case of the attack against memory traces; this is captured in histogram in Figure 4.3. However, this surprisingly did not happen when attacking outputted values, see Note 4.7. We did not find any explanation for this.

Conclusion

After a general introduction to cryptography, we presented three attack contexts: black-box, gray-box and white-box, with an emphasis on the last-named. Then we described another pillar of this thesis – the Advanced Encryption Standard (AES, [47]) and its white-box variant by Chow et al. (WBAES, [16]).

Next we presented a side-channel attack that seems to be unrelated to the white-box attack context. However, we also described their relationship as originally outlined by Bos et al. [12], and provided a detailed description of this novel attack. Here we particularly emphasized one practical consequence – no reverse engineering effort is needed for this attack.

In Chapter 3, we first reproduced results of the attack by Bos et al. against Klinec's implementation [29] of WBAES. Then we extended this attack from 16 to 255 targets, and finally we introduced a simple unifying approach for our contributions.

In the fourth chapter, we analyzed the attack from three perspectives. Firstly, we suggested a blind attack (i.e., an attack without knowing the actual key) based on our observations. Secondly, we commented on the usage of the attack in a white-box cryptography design and presented possible practical countermeasures that, however, did not seem to avoid this attack. Thirdly, we analytically identified and experimentally confirmed which intermediate result leaks information. We also outlined which is the only building block of WBAES that attempts to protect it from our enhanced attack, and why its choice is inappropriate.

We concluded this thesis in Chapter 5, where we summarized possible topics of further research.

References

[1] Side-Channel Marvels. Git repositories. `https://github.com/SideChannelMarvels`.

[2] D Agrawal, B Archambeault, J Rao, and P Rohatgi. The EM Side-Channel (s): Attacks and Assessment Methodologies. *Internet Security Group, IBM Watson Research Center. ps*, 2(3), 2002.

[3] Dmitri Asonov and Rakesh Agrawal. Keyboard acoustic emanations. In *Proceeding of the IEEE Symposium on Security and Privacy 2004*, pages 3–11. IEEE, 2004.

[4] American Bankers Association et al. Triple Data Encryption Algorithm Modes of Operation. *ANSI X9*, pages 52–1998, 1998.

[5] Kerckhoffs Auguste. La Cryptographie Militaire. *Journal des sciences militaires*, 9:538, 1883.

[6] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In *Advances in Cryptology–EUROCRYPT 2014*, pages 221–238. Springer, 2014.

[7] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Advances in cryptology—CRYPTO 2001*, pages 1–18. Springer, 2001.

[8] Elad Barkan and Eli Biham. In how many ways can you write Rijndael? In *Advances in Cryptology–ASIACRYPT 2002*, pages 160–175. Springer, 2002.

[9] Lenka Bačinská. White-box attack resistant cipher based on WBAES. 2015.

[10] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *Selected Areas in Cryptography*, pages 227–240. Springer, 2004.

[11] Alex Biryukov, Christophe De Canniere, An Braeken, and Bart Preneel. A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In *Advances in Cryptology—EUROCRYPT 2003*, pages 33–50. Springer, 2003.

[12] Joppe W Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential Computation Analysis: Hiding your White-Box Designs is Not Enough. Technical report, Cryptology ePrint Archive, Report 2015/753. `http://eprint.iacr.org/2015/753`, 2015.

[13] Zvika Brakerski and Guy N Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In *Theory of Cryptography*, pages 1–25. Springer, 2014.

[14] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology—CRYPTO'99*, pages 398–412. Springer, 1999.

[15] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C Van Oorschot. A white-box des implementation for drm applications. In *Digital Rights Management*, pages 1–15. Springer, 2002.

[16] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, pages 250–270. Springer, 2002.

[17] Matt Curtin and Justin Dolske. A brute force search of des keyspace. In *8th Usenix Symposium, January*, pages 26–29. Citeseer, 1998.

[18] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. 1999.

[19] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[20] Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. White-box security notions for symmetric encryption schemes. In *Selected Areas in Cryptography–SAC 2013*, pages 247–264. Springer, 2013.

[21] Tim Dierks. The transport layer security (TLS) protocol version 1.2. Online: `https://tools.ietf.org/html/rfc5246`, 2008. Accessed: 2015-03-06.

[22] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems—CHES 2001*, pages 251–261. Springer, 2001.

[23] Dan Goodin. Crack in Internet's foundation of trust allows HTTPS session hijacking. `http://arstechnica.com/security/2012/09/crime-hijacks-https-sessions/`. Accessed: 2015-03-06.

[24] Louis Goubin and Jacques Patarin. DES and differential power analysis the "duplication" method. In *Cryptographic Hardware and Embedded Systems*, pages 158–172. Springer, 1999.

[25] NewAE Technology Inc. ChipWhisperer™ Project. `http://newae.com/chipwhisperer`. Accessed: 2016-02-07.

[26] Matthias Jacob, Dan Boneh, and Edward Felten. Attacking an obfuscated cipher by injecting faults. In *Digital Rights Management*, pages 16–31. Springer, 2002.

[27] Pascal Junod. On the complexity of Matsui's attack. In *Selected Areas in Cryptography*, pages 199–211. Springer, 2001.

[28] Mohamed Karroumi. Protecting white-box aes with dual ciphers. In *Information Security and Cryptology-ICISC 2010*, pages 278–291. Springer, 2010.

[29] D. Klinec. Whitebox-crypto-AES. Git repository. `https://github.com/ph4r05/Whitebox-crypto-AES`.

[30] Dušan Klinec. White-box attack resistant cryptography. 2013.

[31] Ç.K. Koç. *Cryptographic Engineering*. Springer US, 2008.

[32] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer, 1999.

[33] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[34] Markus G Kuhn. Optical time-domain eavesdropping risks of CRT displays. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 3–18. IEEE, 2002.

[35] Joe Loughry and David A Umphress. Information leakage from optical emanations. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):262–289, 2002.

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[37] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology—EUROCRYPT'93*, pages 386–397. Springer, 1993.

[38] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[39] Wil Michiels, Paul Gorissen, and Henk DL Hollmann. Cryptanalysis of a generic class of white-box implementations. In *Selected Areas in Cryptography*, pages 414–428. Springer, 2008.

[40] Wilhelmus Petrus Adrianus Johannus Michiels and Paulus Mathias Hubertus Mechtildis Antonius Gorissen. Cryptographic method for a white-box implementation, November 9 2007. US Patent App. 12/514,922.

[41] James A Muir. A Tutorial on White-box AES. Technical report, Cryptology ePrint Archive, Report 2013/104. `http://eprint.iacr.org/2013/104`, 2013.

[42] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[43] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security*, pages 529–545. Springer, 2006.

[44] Colin O'Flynn. Example Captures. `https://www.assembla.com/spaces/chipwhisperer/wiki/Example_Captures`. Accessed: 2016-02-07.

[45] Fabien Petitcolas. Kerckhoffs – Cryptographie militaire. `http://www.petitcolas.net/kerckhoffs/#english`. Accessed: 2016-02-03.

[46] NIST FIPS PUB. 46-3: Data Encryption Standard. *Federal Information Processing Standards, National Bureau of Standards, US Department of Commerce*, 1977.

[47] NIST FIPS PUB. 197: Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication*, 197:441–0311, 2001.

[48] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Smart Card Programming and Security*, pages 200–210. Springer, 2001.

[49] Håvard Raddum. More dual rijndaels. In *Advanced Encryption Standard–AES*, pages 142–147. Springer, 2004.

[50] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[51] Claude E Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949.

[52] Claude E Shannon and Warren Weaver. The mathematical theory of communication. *University of Illinois Press*, 1949.

[53] Philippe Teuwen. MoVfuscator Writeup. `http://wiki.yobi.be/index.php?title=MoVfuscator_Writeup&oldid=9714`. Accessed: 2015-11-03.

[54] Philippe Teuwen. NSC Writeups. `http://wiki.yobi.be/index.php?title=NSC_Writeups&oldid=9629`. Accessed: 2016-03-29.

[55] Yaying Xiao and Xuejia Lai. A secure implementation of white-box AES. In *Computer Science and its Applications, 2009. CSA'09. 2nd International Conference on*, pages 1–6. IEEE, 2009.

Appendix A

Contents of Attached CD

`diploma-thesis.pdf`
: diploma thesis in Portable Document Format,

`attackTools/`
: folder with our attack tools,

  `attack/`
  : our attack toolkit written in Ruby, see Section 3.1,

  `BitwiseDPA/`
  : C++ sources of customized bitwise DPA attack, see Algorithm 2.2,

  `MemoryTracingTools/`
  : C++ sources of memory tracing tools, see Section 2.2.1,

`implementations/`
: folder with (white-box) AES implementations and tables,

  `klinecWBAES/`
  : C++ sources of customized `KlinecWBAES`, see Section 3.2,

  `naiveAES/`
  : C++ sources of `naiveAES`, see Section 3.2,

  `NoSuchCon2013Challenge/`
  : C source of an interesting AES white-box implementation, see Section 5.1,

  `WBAESTables/`
  : our particular 8 instances of `KlinecWBAES` tables, see Section 4.1.1.