uni.lu
UNIVERSITÉ DU
LUXEMBOURG

PhD-FSTC-2013-07
The Faculty of Sciences, Technology and Communication

DISSERTATION

Presented on 05/02/2013 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Jean-François GALLAIS
Born on 10 December 1984 in Coutances (France)

Microarchitectural Side-Channel Attacks

Dissertation defense committee

Dr. Alex Biryukov, vice chairman
Associate professor, Université du Luxembourg

Dr. Jean-Sébastien Coron, chairman
Associate professor, Université du Luxembourg

Dr. Volker Müller, dissertation supervisor
Associate professor, Université du Luxembourg

Dr. Emmanuel Prouff
Agence nationale de la sécurité des systèmes d'information, France

Prof. Dr. François-Xavier Standaert
Professor, Université Catholique de Louvain, Belgique

Abstract

Cryptanalysis is the science which evaluates the security of a *cryptosystem* and detects its weaknesses and flaws. Initially confined to the *black-box model*, where only the input and output data were considered, cryptanalysis is now broadened to the security evaluation of the physical implementation of a cryptosystem. The *implementation* attacks which compose physical cryptanalysis are divided into *fault* attacks, exploiting the effect of disruption of the normal functioning of the device, and *side-channel* attacks, exploiting the dependency between the instructions and data (including key bits) processed by a device and its physical characteristics (e.g. execution time, power consumption, electromagnetic (EM) radiations). In the scope of this thesis, we particularly focus on the latter attacks.

"Every computation leaks information" and lowering the physical leakages of an implementation is indeed a complex task both from cryptographic and engineering viewpoints, especially when performance and cost enter the equation. The development of adequate countermeasures necessitates a thorough knowledge of the various vulnerabilities that the microcontroller induces. Although generic side-channel attacks such as Differential Power Analysis (DPA) can generally retrieve the key with weak assumptions on a cryptographic implementation, we show in this thesis that the focus on specific components and properties from the architecture of the target device may allow an adversary to yield better success in a key recovery and sometimes to thwart DPA countermeasures.

First, we elaborate on attacks which deduce the cache activity of a device from single side-channel traces and algebraically exploit this information to recover the key. We propose different attacks against embedded software implementations of the Advanced Encryption Standard (AES) in the chosen- and known-plaintext scenarios and make them tolerant to environments where high noise or a partially preloaded cache would normally introduce errors in the key recovery. Second, we discuss the failure of standard DPA against the modular addition and propose a practical and generic approach to circumvent it.

Third, we show that microarchitectural leakages and fault inductions can be exploited in a constructive way when induced by hardware Trojans implemented on general-purpose microprocessors. Such Trojans can either provide an adversary with a backdoor access to the trojanized device executing an arbitrary cryptographic software or serve to protect the Intellectual Property (IP) of the chip designer through digital watermarking.

The last part concerns divide and conquer side-channel attacks such as DPA. Testing different combinations of key chunk candidates turns out to be very complex when the individual chunk recoveries are bounded in measurement complexity or performed in noisy environments. We address the so-called key enumeration problem with an efficient sorting method.

Résumé

La cryptanalyse a pour but d'évaluer la sécurité d'un *cryptosystème* et de déceler ses failles. D'abord cantonnée au modèle dit *en boîte noire*, où sont seulement considérées les données en entrée et en sortie de l'algorithme, la cryptanalyse prend maintenant également en compte l'implémentation physique d'un cryptosystème. Les attaques d'*implémentation* qui constituent la cryptanalyse physique sont composées d'une part des attaques par *faute* (exploitant les effets d'une perturbation du fonctionnement de l'appareil) et d'autre part des attaques par *canal auxiliaire* (exploitant les dépendances entre les instructions et données (dont la clé) traitées par l'appareil et ses caractéristiques physiques (telles que le temps d'exécution, la consommation électrique ou les émanations électromagnétiques). Dans le cadre de cette thèse, nous nous intéressons particulièrement à ces dernières.

"Chaque calcul laisse fuir de l'information" et atténuer les fuites d'une implémentation est en effet une tâche complexe tant d'un point de vue algorithmique que technique, en particulier quand performance et coût entrent en considération. La conception de contre-mesures adéquates nécessite une profonde connaissance des diverses failles qu'un microcontrôleur peut induire. Bien que les attaques génériques par canal auxiliaire permettent généralement de retrouver la clé avec peu de connaissances sur l'implémentation cryptographique visée, nous montrons dans cette thèse que certains composants de l'architecture de l'appareil peuvent parfois permettre à un adversaire de retrouver plus efficacement la clé et de contourner des contre-mesures génériques.

En premier lieu, nous traitons des attaques qui déterminent l'activité du cache d'un microprocesseur à partir de simples courbes de mesures et qui exploitent cette information pour retrouver la clé. Nous proposons différentes attaques à message choisi et à message connu sur des implémentations logicielles embarquées d'AES et les adaptons à la présence de bruit dans les mesures ou au préchargement partiel du cache. Ensuite nous abordons l'échec des attaques standard par analyse différentielle de la consommation sur les additions modulaires et proposons une approche générique et pratique pour y remédier.

D'un angle de vue différent, nous montrons que les fuites microarchitecturales ainsi que les fautes peuvent être exploitées de façon constructive quand elles sont induites par des chevaux de Troie physiques implémentés sur des microprocesseurs. Ces chevaux de Troie peuvent fournir à un adversaire une porte dérobée vers les clés secrètes de l'appareil ou bien protéger la propriété intellectuelle du concepteur de la puce par un tatouage numérique.

La dernière partie concerne les attaques par canal auxiliaire *diviser pour régner*. Trouver la bonne combinaison à partir de différents candidats pour chaque morceau de clé peut s'avérer difficile voire impossible, en particulier quand le nombre de mesures est limité ou quand les mesures comportent beaucoup de bruit. Nous proposons une méthode efficace d'énumération des clés en solution à ce problème.

Acknowledgements

I would like to thank Alex Biryukov, Jean-Sébastien Coron and Volker Müller, my supervisor, who offered me the opportunity to undertake my doctoral studies and advised me all along this four year adventure. I especially thank Volker for the improvements he suggested to this manuscript. I am honoured and grateful to Emmanuel Prouff and François-Xavier Standaert for serving in the thesis jury.

I am indebted to the persons who introduced me to cryptology and encouraged me to pursue this fabulous field of science: Fabien Laguillaumie, Marc Girault, Pascal Paillier and Elisabeth Oswald.

I express my profound gratitude to Ilya Kizhvatov for imparting me his knowledge and experience in scientific research in cryptology. Our countless discussions were a great source of inspiration.

I am thankful to Arnab Roy and Praveen Kumar Vadnala for our effective collaboration and discussions. I thank my co-authors Johann Großschädl, Neil Hanley, Markus Kasper, Marcel Medwed, Francesco Regazzoni, Jörn-Marc Schmidt, Stefan Tillich, Michael Tunstall, Marcin Wójcik and the VAM2 ECRYPT research group for our motivating and highly interesting discussions. I deeply thank Andrey Bogdanov for our stimulating collaboration.

It was a great pleasure to share the office with Avradip Mandal and exchange with him about scientific and less scientific issues. My many thanks go to my colleagues from the Laboratory of Algorithmics, Cryptology and Security of the University of Luxembourg for the friendly yet working atmosphere: David Galindo, Dmitry Khovratovich, Yann Le Corre, Gaëtan Leurent, Zhe Liu, Ivica Nikolić, Deike Priemuth-Schmid, Ivan Pustogarov, Vesselyn Velichkov, Srinivas Vivek Venkatesh, Ralf-Philipp Weinmann and Bin Zhang.

∴

Je remercie mes amis du Luxembourg pour tous les bons moments partagés ainsi qu'Iván Boumans et les musiciens de l'ensemble instrumental de l'université, qui ont permis que la musique m'accompagne tout au long de cette aventure.

Je remercie mes amis normands pour tous nos fabuleux voyages et formidables soirées passés et à venir! et pour le bonheur que nous avons à nous retrouver malgré les distances.

Je remercie Uršula pour son amour, son soutien au quotidien et sa patience dans les moments les moins faciles.

Je remercie mes chers parents pour les encouragements qu'ils m'ont toujours apportés. Je remercie enfin ma soeur et mon frère et leur redis toute la place qu'ils tiennent dans mon coeur.

Jean-François Gallais, Décembre 2012

Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

AES	Advanced Encryption Standard	iii
ASIC	Application-specific integrated circuit	5
CMOS	Complementary Metal–Oxide–Semiconductor	8
CNF	Conjunctive Normal Form	7
CPU	Central Processing Unit	xv
CRT	Chinese Remainder Theorem	68
DES	Data Encryption Standard	4
DPA	Differential Power Analysis	iii
DRM	Digital Rights Management	3
DSO	Digital Sampling Oscilloscope	19
EM	electromagnetic	iii
FIPS	Federal Information Processing Standard	4
FPGA	Field-Programmable Gate Array	5
IP	Intellectual Property	iii
LSBi	least significant bit	47
LSBy	least significant byte	75
LTOR	left to right	68
MAC	Message Authentication Code	3
MAM	Memory Accelerator Module	19
MSBi	most significant bit	47
MSBy	most significant byte	69
NESSIE	New European Schemes for Signatures, Integrity and Encryption	47
NIST	National Institute of Standards and Technology	5
PMF	Probability Mass Function	xiii
RTOL	right to left	68
SEMA	Simple Electromagnetic Analysis	6
SHA	Secure Hashing Algorithm	5

SPA	Simple Power Analysis	6
TC	Trusted Computing	64
TPM	Trusted Platform Module	64
TSC	Trojan Side-Channel	64
VPN	Virtual Private Network	3
XOR	exclusive-or	xiii

Chapter 1

Introduction

More than a decade ago, side-channel attacks emerged, threatening the security of embedded secure systems. They exploit physical measurements—during normal functioning of the device and without requiring any alteration of it—so as to retrieve the secret key involved; they require less calls to the target implementation than classical cryptanalysis generally would. This thesis deals with these attacks and information leakages emanating from microarchitectural features of the target device.

Contents

1.1 The art of secret writing

Since time immemorial, man has been protecting his valuable possessions with doors, locks and safes. Likewise, he has also been needing to prevent the acquaintance with important written messages from undesired readers. Whereas restrained access to a message could be ensured through physical protection or by information hiding[1], confidentiality had sometimes to be achieved via a code (i.e. word substitution onto

[1] The art of hiding information, called steganography, is as ancient as cryptography and has been used in parallel or in addition to it in order not to attract attention. Steganography is nowadays also part of information technology, providing techniques for covert communication or digital watermarking.

a different alphabet) or a cipher (i.e. letter substitution onto the same alphabet). From Julius Caesar announcing his general who was besieged his imminent rescue to Marie Stuart fomenting with her partisans her own escape from prison and the assassination of her cousin Elisabeth I of England, from the foiled German assault on Paris in June 1918 during the First World War to Alan Turing and his peers breaking the code of the most complex cipher machine ever built—the famous Enigma—thus precipitating the collapse of the Third Reich, the history of mankind abounds with episodes in which cryptography and cryptanalysis—its “destructive” counterpart—played a crucial role in the course of events.

1.1.1 Terminology and elementary concepts

In the simple scenario where two entities want to communicate without any third party being able to understand the meaning of their exchange, cryptography aims to ensure confidentiality, that is, it provides an *encryption* technique (parametrized with a *secret key*) that the sender, Alice, will apply to her *plaintext* to obtain a *ciphertext*. The ciphertext can be sent to Bob through any *unsecure* channel on which an eavesdropper would be possibly acting. Bob, the legitimate receiver and as such, in possession of the secret key, applies upon receipt the inverse mapping of the encryption, i.e. the *decryption* routine, also parametrized by the secret key, so as to retrieve the plaintext.

Ciphering techniques have evolved at a slow pace through the ages [104, 108]. The recurrent idea was to map the plain alphabet onto one enciphered alphabet (e.g. Caesar’s cipher, which was formally broken with the development of frequency analysis of the letters in the ciphertext by Al-Kindi in the 9th century [5]), then onto several ones with polyalphabetic ciphers (e.g. Vigenère cipher in the 15th century, more resistant to frequency analysis yet broken in the 19th century). The key determinates which permutations of the alphabet are to be used for each letter of the ciphertext (resp. plaintext).

The successful *cryptanalysis* of the Vigenère cipher and its variants shed light on the necessity to rethink the very foundations of cryptography so as to come up with truly secure and easy to use ciphers, at a period of time where telegraphy was growing in popularity and the use of weak ciphers had already been proven to be devastating during wars. In 1883, Kerckhoffs described such fundamental principles [63]. We can retain these three:

1. the cipher must be practically, if not mathematically, secure.

2. the cipher structure and all details but the key can be disclosed without jeopardizing the cipher security.

3. the system must be easy to use and should not require mental skills nor the knowledge of complex rules from the user.

These principles are generally still in use in the conception of modern ciphers and other cryptographic algorithms and systems.

1.2 The science of confidentiality and trust

Starting in 1949 with Shannon's information theory and the introduction of key concepts like *entropy* (a quantification for information) and algorithmic complexity [103], the modern era of cryptography is of high contrast with its dark age.

First, the wide deployment of computers and their high computing power enable the execution of complex cryptographic algorithms in a fraction of a second without requiring any skills from the user (which satisfies one of Kerckhoffs' principles (3.)). In the meantime, this high power is also available to the adversary, and Internet and the rise of cloud computing increase complexity boundaries all the more that it is hard to estimate what the adversary computing capabilities exactly are.

Second, from the art of secret writing, cryptography evolved to the science of confidentiality and trust, ensuring more properties to electronic communications than the sole message secrecy. Cryptography also provides:

- **data integrity:** Accidental errors during transmission may be detected and corrected with error-correcting codes, but not deliberate alteration of the message by a third-party, because error-correcting codes are not keyed algorithms. However, cryptography develops *signature* schemes which produce a signature based on the message content and the sender's key, and Message Authentication Codes (MACs), which produce a *tag* based on the message content and the sender's key. The validity of a signature or a tag, thus the message integrity, can be verified by the receiver.

- **authentication:** In the same way as for data integrity, the verification of a signature or a tag allows the receiver of a message to verify the message origin.

- **non-repudiation:** Cryptography ensures that the sender of a message cannot later deny the message origin because only her has the ability to produce a valid signature.

Third, the use of cryptography is not anymore confined to diplomatic and military usage, but is now part of everyone's everyday life. It is present in mobile phones, smart cards, access tokens, car keys, biometric passports, Internet secure web browsing, Virtual Private Networks (VPNs), various media supports for Digital Rights Management (DRM) and many more applications. The purpose of modern cryptography is to facilitate electronic transactions, access and key managements (e.g. in complex logistic infrastructures) and ensure privacy of communications and authentication of users. A more open world as ours means more connections between nodes (natural persons or corporate bodies). Cryptography provides the modern "locks" and "seals", i.e. the confidentiality and trust that our communications require.

The fourth difference between ancient and modern cryptography is directly related to the scope of this thesis. Since more than a decade, the security assessment of a system has gone beyond the *black-box model*, where only plaintexts and ciphertexts, that is, the inputs and outputs of the cipher were taken into account so as to find the cipher weaknesses. The seminal work of Kocher [65] and many more publications afterwards have shown that other information sources are available to an adversary if she considers the entire system instead of the cryptographic algorithm alone. Going from a mathematical description of the cipher to the physical observation of its implementation, more information channels appear: the execution timing, the power consumption and the EM emanations are the most investigated *side-channels*, but other ones may also be exploitable, e.g. photonic [96] and acoustic emissions [55,102], thermal variations [28].

Encryption and authentication schemes divide into two classes: symmetric and asymmetric cryptographies.

1.2.1 Symmetric cryptography

In symmetric cryptography, the sender Alice and the receiver Bob share the *same* secret key. Alice uses the key to encrypt the message, and Bob uses the key to decrypt it. The MAC serves as a proof for Bob that the sender is in possession of the secret key and that the message has not been *forged* nor altered.

The first kind of symmetric encryption techniques are block ciphers, where a deterministic algorithm, parametrized by the key, is applied to a fixed-length block of plaintext bits. Widely used examples of block ciphers include the Data Encryption Standard (DES) [41] and AES [42], announced resp. in 1976 and 2001 as Federal Information Processing Standards (FIPSs). We describe the AES in Section 2.2.5. Not addressed in this thesis, the other design for symmetric encryption are the stream ciphers, where a pseudorandom bit stream is combined with the plaintext bits.

1.2.2 Asymmetric cryptography

In 1976 and 1977, two major breakthroughs were made in the history of modern cryptography and correspond to the emergence of asymmetric cryptography.

First, Diffie and Hellman invented the first key exchange protocol: assume two parties have each their own secret key, it allows them to agree on a common secret key without revealing each other their own secret key [38]. The commonly produced key can then be used within a symmetric encryption scheme.

The second milestone was the development of RSA, the first asymmetric encryption scheme, named after their authors: Rivest, Shamir and Adleman [95]. In this paragraph, we use RSA as a simple illustration of public-key cryptography. In RSA, a public modulus N is defined as the product of two large primes p and q which ought to be kept secret. A user is provided with a public key e and a private key d, the latter is computed as the inverse of e modulo $\varphi(N)$, where φ is Euler's totient function. When Alice wants to encrypt a plaintext m intended for Bob, she encrypts it using Bob's public key and obtains a ciphertext c in this way: $c = m^e \pmod{N}$. Upon receipt of c, Bob decrypts it using his private key d to retrieve m: $m = c^d \pmod{N} = (m^e)^d \pmod{N} = m^{ed} \pmod{N} \equiv m \pmod{N}$, using Fermat's little theorem since $ed \equiv 1 \pmod{\varphi(N)}$. Otherwise described, a public key is a "lock" that anyone can use to protect a message; on the contrary, only the private key opens the lock, therefore only the intended receiver of the message can open it. The RSA signature works in the converse way: Alice signs the message using her own private key and then, using Alice's public key (which is available to all users including Bob), Bob verifies a signature that, if valid, can only have been produced with Alice's private key.

To date, RSA is still the most widely used algorithm for asymmetric encryption and signatures.

1.2.3 Hash functions

In order to enhance both the security and performance of encryption and authentication routines (symmetric and asymmetric), cryptographic tools like hash functions are required. From an input message of any length, these functions output a fixed length *hash*. Whereas the hash function must be very fast to execute, it must be practically infeasible to invert it, that is, to retrieve the input message from the hash (preimage resistance). Other properties are desired in hash functions: from a given message and its hash, it must be practically impossible to find another message which has the same hash (second-preimage resistance); and also, it should be practically impossible to find two messages whose hashes are the same (collision resistance).

Hash functions are used in the generation of MACs, signatures and fingerprints, for data integrity. The most widely used hash functions are the ones from the Secure Hashing Algorithm (SHA) family: SHA-1 and SHA-2 [43] (although the former is vulnerable to collision attacks). The U.S. National Institute of Standards and Technology (NIST) recently launched the SHA-3 competition where worldwide submissions were analysed in an open process by the cryptographic community (thus satisfying one of Kierckhoffs' principles (2.)). The hash function Keccak [13] has been selected as the SHA-3 algorithm, but the hash function Skein [40], that serves as a case study in Chapter 3, was one of the five finalists out of 51 candidates.

1.3 Side-channel analysis

In classical cryptanalysis, the inputs of a key recovery are functions of the input and output data of the cryptographic algorithm. In physical cryptanalysis however, the adversary also take into account sources of information emanating from the physical implementation of the algorithm.

A cryptographic algorithm is a sequence of mathematical operations performed on plaintexts or ciphertexts and parametrized by the key[2]. As cryptography is not anymore performed with paper and pencil but with electronic integrated circuits (general-purpose microprocessors, Field-Programmable Gate Arrays (FPGAs) or Application-specific integrated circuits (ASICs)), the system aimed at executing the cryptographic steps consumes power, emanates electromagnetic radiations and takes a certain amount of time to complete the code execution. These characteristics among others, which are physically observable and measurable, depend on the

[2] Hash functions are not keyed algorithms, but a side-channel attack against a hash function can take place for example against a MAC construction or against the physical random number generator used by the hash function.

architecture details of the device, the instructions performed as well as the data processed, which includes the secret key. As a consequence, based on assumptions on the way the device behaves and leaks information through a side-channel, the adversary can try to exploit the dependencies between the measurements and the key value in play in order to gain information about the key.

The power consumption and the EM emanations are the most investigated and powerful side-channel because they offer a more fine-grained observation of the device activity than the timing (considered in Chapter 4), i.e. several points of time can be independently observed and analysed. The EM side-channel (Chapter 2) is an indirect measure of the power consumption through its EM field.

Side-channel analysis forms a diverse field of attacks which have in common to be *passive* (the normal functioning of the device is not disrupted, as opposed to fault attacks) and *non-invasive* (the device is not damaged during the attack, as opposed to certain fault attacks, e.g. using a laser beam or requiring decapsulation of the chip).

Simple side-channel analysis (Simple Power Analysis (SPA) or Simple Electromagnetic Analysis (SEMA)) is a direct interpretation on the key material used by the target device from one or few power traces while it performs cryptographic executions [66]. Indeed, when the execution of an instruction set is conditioned or ordered by bits of the key, the different power patterns indicate information on the key itself. Note that SPA and SEMA attacks require a detailed knowledge on the algorithm implementation. Specific care has to be taken by a designer to prevent these attacks [32]. On a different issue, simple side-channel analysis also helps a reverse-engineer to get knowledge about the algorithm run by the device and the details of its implementation. We provide in Chapter 2 an example of these attacks.

The classification we adopt for side-channel attacks is twofold: (1) the attacks which build equations involving key bits from the side-channel traces and solve them, which we denote *analytic* attacks; and (2) the attacks which recover the key chunk by chunk, which we denote *divide and conquer* attacks.

1.3.1 Analytic attacks

Analytic side-channel attacks draw algebraic equations involving the key bits from the observation or measurements of the side-channel, typically the power consumption or the EM emanations. Otherwise described, the side-channel leakage induces constraints on the key variables which allow to reduce the key space to a feasible size for an exhaustive search. When the number of constraints and key variables is large, SAT solvers can be applied.

A first example of analytic attacks are *microarchitectural attacks* [2, 3], which exploit particular components or features of the processor such as data cache (as described in Chapter 2), instruction cache or functional units (e.g. multipliers, as described by Großschädl et al. [53] and in Chapter 4).

Collision attacks are more generic, that is, they have a smaller dependence to the device architecture. They detect from the side-channel trace the similarities of leakage when equal values are processed and allow an adversary to draw equalities from these values involving key bits [19–21,99,100].

Algebraic attacks first describe the block cipher with a system of equations in Conjunctive Normal Form (CNF) with bit variables. In a second—online—stage, this satisfiability problem is resolved with the additional information provided by the side-channel measurements [92,93,124]. They typically have a low measurement complexity and can easily deal with masking countermeasures. However, they are not robust and heavily rely on the leakage model and the quality of the measurements.

1.3.2 Divide and conquer attacks

Divide and conquer side-channel attacks, with the notable example of DPA, take advantage of the architectural and design constraints which make intermediate values have a sufficiently low size to allow an adversary to examine all *subkey* hypothesis—this is the *divide* stage, addressed in Chapter 3 against modular addition. The other subkeys which by concatenation form the *full* key are recovered in the same manner—this is the *conquer* stage, on which we bring a particular attention in Chapter 5.

In the following, we give a brief overview of SPA and DPA attacks, using the power consumption as the side-channel.

Differential power analysis

Inspired by the algorithmic description of DPA by Mangard et al. [76], a DPA attack can be described with the following steps:

0. **Device profiling (optional step)**

   A DPA attack can be made more efficient if the device leakage is profiled [31]. Suppose that the adversary possesses a similar device as the one under attack; she can record the power consumption of the device for each subkey and each data values and compute a template for each such pair. The templates are later used for statistical comparison in Step 5 with the leakage from the device under attack. A power model (Step 4) can be applied so as to further reduce the number of templates to build.

1. **Choice of target function**

   An intermediate value of the cipher implemented on the device under attack, which has to be a function of key and data (plaintext or ciphertext) bits, is chosen by the adversary. This intermediate value should be carefully chosen, so that the offline complexity of the attack does not exceed the computing capabilities of the adversary. On the other hand, the higher the non-linearity of the target function, the better the success of the attack, as shown by Prouff [88]. In Chapter 3, we discuss the impact that the choice of a target function has in the success of a DPA attack.

2. **Acquisition of power consumption**

   From the control interface of the device (e.g. a Personal Computer (PC)), the adversary submits input data to the implementation and, if needed in the attack, obtains in return the output data. When possible for the adversary, she enables a trigger signal around the target operation. This will ensure that the power traces are correctly aligned, that is, each point of time corresponds to the same executed instruction in all recorded traces. The power measurements are obtained via a measurement setup, as depicted in Figure 1.1.

3. **Hypothetical intermediate values building**

   Based on the target function, the adversary builds hypothesis for the intermediate value for each subkey hypothesis and input data.

4. **Power modelling**

   In this step (possibly omitted in a profiled attack), the intermediate values are mapped to hypothetical power consumption values. It requires the choice of an adequate power model. For example, one can assume that the dynamic power consumption of a Complementary Metal–Oxide–Semiconductor (CMOS) circuit is proportional to the number of set bits in the processed value, since a set bit output by a cell requires this cell to be active, thus having a dynamic power consumption; on the contrary, an unset bit indicates that the cell is inactive, and that its power consumption is dropped to its static part. Therefore, the power consumption of a processed value can be modelled as its Hamming weight. The Hamming distance, taken between two states of a register (Chapter 3), is also a common choice of power model since it captures the transitions of state, which determine the dynamic power consumption of a circuit.

5. **Statistical comparison**

   In order to recover the subkey used by the device, the final step is to compare the power traces acquired in Step 2 with the hypothetical power consumption values obtained in Step 4 for different subkey candidates. Every time position of the power traces is considered, and for the positions at which the target intermediate value is processed, a strong relation to the hypothetical power consumption values is expected. Common comparison functions are Pearson's correlation coefficient [27] and Kocher's difference of means [66].

1.4 Contributions

In this thesis, we elaborate on side-channel attacks against software implementations of cryptographic algorithms and the microarchitectural leakages induced by the device architecture. The contributions are presented in chapters as follows:

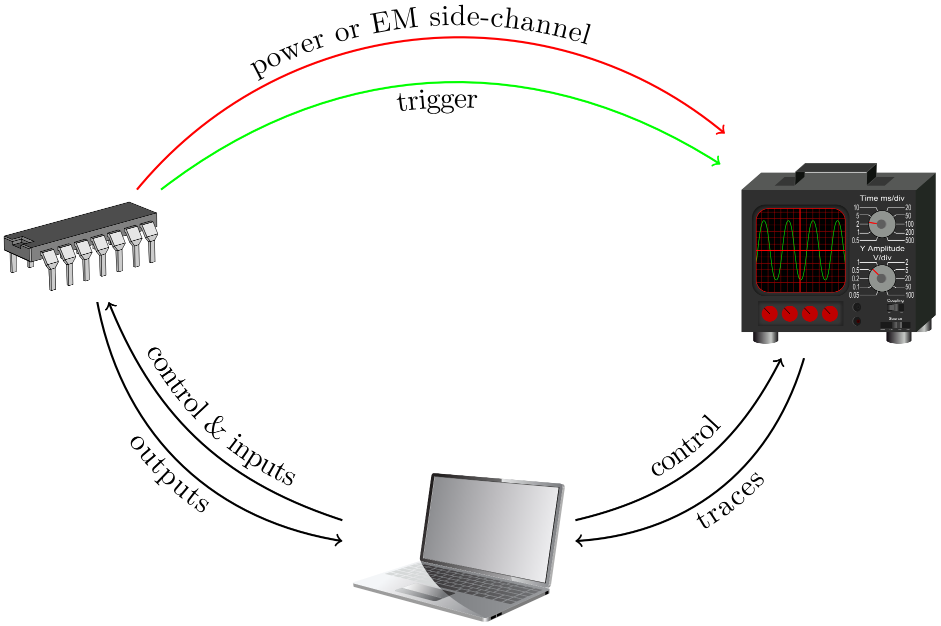


Figure 1.1: Diagram of a typical measurement setup for power or EM analysis attack

- In Chapter 2 we show that the activity of the cache memory of embedded processors is observable on EM traces with experiments conducted on a 32-bit ARM microcontroller. We develop efficient key recovery algorithms exploiting the cache activity gained from power or EM measurements against symmetric encryption algorithms such as AES, when implemented in software with lookup tables. Our attacks can stand the presence of noise in the measurements (while previous works were assuming perfect cache event detection) and the preloading of AES data in the cache. We present a theoretical model for the known plaintext attack and we confirm its soundness with simulations. We discuss the relevance of common countermeasures against DPA in the scope of trace-driven cache-collision attacks. The chapter is based on publications at WISA and COSADE [47, 48].

- In Chapter 3 we elaborate on DPA attacks against a modular addition. The carry bits it produces make standard DPA attacks fail. While a more sophisticated distinguisher has recently been proposed to circumvent this problem in the Hamming weight power model, we focus on the target function and experiment the recovery of key bits involved in a combination of two modular additions. This practical approach is more generic with respect to the power model and successfully recovers the key bits, at the expense of the number of key hypothesis to deal with. To keep a feasible attack complexity, we put forward a divide and conquer strategy. We verify the success of our methods against a high performance implementation of the block cipher Threefish on an 8-bit AVR microcontroller. The chapter is based on a publication at

WESS [115].

- In Chapter 4 we introduce microarchitectural Trojans, which are tiny modifications of the hardware design of a processor so as to induce or amplify a microarchitectural leakage or to insert a computational fault during the execution of cryptographic algorithms. Such hardware Trojans can serve as a backdoor for key recovery, while remaining undetected during normal functioning of the device. We discuss the scenarios for a Trojan to be inserted by a chip designer or manufacturer and we propose novel software-based activation mechanisms, which renders the detection of a Trojan practically infeasible. The disclosure of the key can take place through existing fault or side-channel attacks. We describe two realistic scenarios where an attacker would be able to recover the AES secret key and the RSA private key used by an OpenSSL software. The chapter is based on a publication at INTRUST [46].

- In Chapter 5 we propose an algorithm for combining and sorting full key hypothesis according to the PMF of a single chunk, resulting from a divide and conquer attack such as DPA. It is based on pairwise multiplications of the probability values. We show that an efficient enumeration of the full key candidates speeds up the exhaustive search for the correct key at the end of the attack. Similarly, we show that in a template-based DPA attack, an optimized key enumeration allows an attacker to reduce the complexity of the template-building phase.

Chapter 2

Side-channel trace-driven cache-collision attacks

In this chapter we elaborate on the exploitation of information induced by the cache mechanism and directly observable in a side-channel trace. We verify in practice on a 32-bit ARM microcontroller that different cache events are distinguishable on an EM trace. We describe in detail key recovery algorithms for embedded AES software implementations and we show that it is possible to retrieve the 128 bits of the key within 30 encryptions and a negligible offline computation on a standard PC. Our attacks can stand the presence of high amounts of noise in the measurements as well as the partial pre-loading of the cache with AES data. Furthermore, we review several countermeasures in the scope of trace-driven cache-collision attacks and we show that special care has to be taken in their implementations. The measurement complexity of our attacks is theoretically estimated and a univariate model is improved. Meanwhile, we show that only a multivariate model is sound.

This is a joint work with Ilya Kizhvatov and Michael Tunstall, published in the proceedings of WISA 2010 [48] and with Ilya Kizhvatov, published in the proceedings of COSADE 2011 [47].

Contents

2.1 Introduction

Among the microarchitectural mechanisms a microprocessor may feature, caching is one of the most common ones. It aims at increasing the speed at which instructions are executed, by storing in a small but fast volatile memory data and instructions which are susceptible of being required in the computation process in a close future. Without the use of a cache memory, the processor has to wait for the required data to be fetched from the (typically slow) non-volatile memory into the registers. This latency represents a bottleneck in the computation flow. Not only the time of execution is increased, but also a superior amount of energy is involved in the operation.

On another issue, it has been noted that on some cryptographic implementations, the execution time varies depending on the instructions and data processed, which possibly allows an attacker to gain secret information while measuring the time taken by an operation where the secret is involved [65]. Similarly, power analysis attacks aim at recovering secret information processed during a computation through the observation of the power consumption of the device under attack [27, 66]. The electromagnetic radiations emitted by the device have also shown a dependency with the instructions executed and the data manipulated, potentially exploitable by an adversary [49, 90]. In the scope of this thesis, one can legitimately ask whether caching also induces dependency between the secret data and the physically observable characteristics of the microprocessor, and can caching be seen as a microarchitectural side-channel.

The affirmative answer was first given a decade ago by Page [85]. Since then, more evidence has been given with the devising of various ways for exploiting caching as a microarchitectural side-channel, and different kinds of cache-collision attacks have been put forward. Their applicable scenarios range from power analysis on embedded devices to timing attacks on remote servers.

In this work, we elaborate on attacks that exploit the knowledge of (possibly chosen) plaintexts with the corresponding side-channel trace (power or electromagnetic trace) of their encryption under a target key. The activity profile of the cache is derived from the observation of single traces: repetition of acquisitions and averaging of the traces are not required, as we later show in Section 2.3, thus our attacks are SPA. Equations involving bits of the key and the plaintexts are deduced from the activity profile. The target key is recovered by the solving of these equations. Such attacks are called *trace-driven* cache-collision attacks, and they also fit into the wider class of side-channel attacks called analytic side-channel attacks.

2.1.1 Our contributions

In this work, we elaborate on cache-collision attacks that analyse the cache activity of a device through the observation of a power or electromagnetic trace. Previous work on these attacks [14, 45] have shown that the deduced equations allow an adversary to gain information on the secret key. The study is presented at the example of the AES-128 block cipher, but the attacks in this work are easy to adapt to the other key lengths and block sizes available for AES.

We start with a description of our practical exploration of the microarchitectural leakage induced by the cache on electromagnetic traces and show that the cache activity of a chip can be easily monitored from a single trace (Section 2.3).

Then, we present a significant improvement to the chosen plaintext attack from Fournier and Tunstall [45] which considerably enables an adversary to reduce the entropy of the key by 60 bits within 14.5 traces instead of 127.5 in the original attack (Section 2.4). Furthermore, we present a known plaintext attack that recovers the entire AES-128 key within less than 30 traces and with a possible exhaustive search over up to 10 key candidates (Section 2.5). This attack follows a similar approach as Acıiçmez and Koç [1] and Bonneau [23], however we treat the case of conventional 256-byte S-box lookup tables (since such tables would be used in a constrained or masked implementation) and precisely describe the full key recovery process. While being comparable to DPA in terms of complexity, our attacks are able to overcome certain countermeasures (Section 2.7).

Our main contribution is the adaptation of both chosen and known plaintext attacks to noisy environments where the detection of the cache events may be erroneous. We also make our key recoveries valid when the cache contains AES data prior to encryption, which would normally cause the attacks to fail. Furthermore, we show with our simulations that the presented attacks tolerate well the presence of these two "real-life" conditions (Section 2.6), requiring a reasonable number of measurements even for significant probabilities of error and amounts of preloaded data.

We discuss the relevance of some countermeasures commonly applied in embedded implementations of block ciphers (Section 2.7). Interestingly, though Boolean masking and random delays are common countermeasures which increase the measurement complexity of DPA attacks to thousands of traces, trace-driven cache-collision attacks naturally defeat some of them without requiring more acquisitions than in the absence of these two countermeasures. The shuffling of the lookups, however, renders trace-driven cache-collision attacks rigorously impossible.

We scrutinize the theoretical model of the attacks and we show that while a univariate model can be used to provide a rough estimation of the attack measurement complexity, the real attack follows a multivariate model (Section 2.8). The results of our simulations confirm the theoretical estimations.

2.2 Generalities

We discuss in this section the role of caching in a microarchitecture. We outline the background of our contribution. We briefly present the AES algorithm used in our study and its important features related to this chapter. We also describe our assumptions about the cache mechanism and detail the notations used through the rest of this chapter.

2.2.1 Caching and performance

For all sorts of microcontrollers, performance has always been a key issue and a never-ending research direction. Even more, for most of the applications rapidity of execution has been the major concern in the development of microprocessors because of the growing demands for computing resources. In the early years of modern computing, other concerns such as low cost design and security were of less importance or simply not considered. To increase the performance of a processor, numerous strategies have been put forward: instructions to be run in parallel; instructions to be pipelined; and also, instructions and data to be stored in a fast and volatile memory: the *cache*.

Driven by research in computing technology, CPUs have increased the speed at which they process code and data. This results in a significantly higher "distance" between the memory and the CPU because the data that the CPU has to process takes relatively more time to be delivered, since the CPU is then waiting for data.

This "distance-based" description of the cache mechanism complies with the average latencies of the different types of memories. The *registers*, which lie within the CPU and form the quickest type of memory available, have an access time of 1 to 3 ns but have a costly design because of their custom CMOS technology, thus allowing only a few bytes of capacity. The *main memory* of a system, which is a DRAM, is on the other hand cheaper but has an access time of 10 to 60 ns. Between

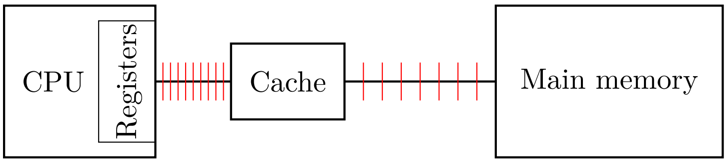


Figure 2.1: “Distances” between the CPU and the memories

the quick, small and costly registers and the slower, cheaper but bigger DRAM, the trade-off on speed, storage capacity and cost is the *cache memory*: it is an SRAM memory, with an average access time comprised within 2 to 12 ns [109]. Caching frequently used data significantly reduces the average latency of a system. For this reason cache memories are present in all PC microprocessors and in a majority of general purpose embedded microprocessors, such as the widespread ARM9 family and the subsequent ARM families [6].

As depicted in Figure 2.1, the cache is a small and fast storage memory that lies between the CPU and the main memory. Every time the CPU requires some data, if first looks for it in the cache. If the data is present, it directly fetches it from the cache and this results in a so-called *cache hit*. If not present, the line of data holding this address is paged from the main memory into the cache and to the CPU registers, because of the assumption that the data around the accessed address is likely to be also accessed in the near future; this results in a *cache miss*. The line of data then remains in the cache until, once full, being overwritten with other lines. Cache hits and misses have different physical characteristics—in particular execution time and power consumption—because of their different latencies and amounts of energy required. These differences are what an adversary exploits in cache-based attacks.

2.2.2 Cache attacks: related work and taxonomy

Following the pioneering articles of Kelsey et al. [62] and Page [85], several notorious attacks have been published involving the cache mechanism and targeting AES. These attacks fall into three types. We outline them in order of increasing means of an adversary as it follows. *Timing-driven* attacks exploit the measured execution time a system takes to run a cryptographic routine. Such a simple measurement setup makes these attacks applicable even on PCs and remote servers. Notable examples are the full key recovery attacks of Bernstein [12] and Bonneau and Mironov [24] against standard and high-performance (i.e. table-driven) implementations of AES requiring the encryption of about 2^{13} to 2^{28} plaintexts. *Access-driven* attacks use a spy process run along the target process, both processes sharing the cache on the system. By using the spy process to clear arbitrary data from the cache or on the contrary to cache chosen data before triggering the target process, and then measuring the corresponding the access time, the adversary can deduce whether this data was used by the target process or not. Applications can be found in both embedded and desktop systems and several attacks have already been published [78, 81]. *Trace-driven* attacks require a fine-grained side-channel, e.g. power consumption or EM emanation, which allows an adversary to distinguish cache hits from cache misses at each lookup. Namely, she is able to produce the sequence of cache events out of one or few side-channel traces. These attacks particularly threaten embedded systems since the latter are exposed to a high risk of power or electromagnetic analysis. In this chapter, we elaborate on this type of attacks.

2.2.3 Cache attacks in the world of side-channel key recovery

Following the comparison of the different types of cache-based attacks, we broaden the scope of study and give a description of trace-driven cache-collision attacks among side-channel attacks.

Side-channel attacks divide into two classes: a) attacks that rely on the resolution of algebraic equations, and b) attacks that rely on a divide and conquer approach. DPA attacks fall in the second class. They aim at recovering for example one or two bytes at a time, computing intermediate values and predicting the power consumption for all key hypothesis, in a total of 2^8 and 2^{16} hypothesis respectively [76].

On the other hand, analytic attacks exploit side-channel leakage to build equations on the key bits. The solving of these equations allows the recovery of the entire key. Notable examples include collision attacks [19, 21, 99, 100], algebraic attacks using SAT solvers [92, 93] and trace-driven cache-collision attacks [1].

2.2.4 Definitions and notations

The four most (resp. least) significant bits of the eight bits of a byte are referred to as its high (resp. low) nibble. The exclusive-or (XOR) operator is defined on inputs of same length such as each output bit is the XOR result on each pair of corresponding input bits. The bitwise XOR operation, stated as "one or the other but not both" is a bitwise addition modulo 2 and has the following truth table:

input		
a	b	output
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1: Truth table of the XOR operator.

Throughout this chapter, we denote:

- the high and low nibbles of a byte b with $\widehat{b}$ and $\widecheck{b}$ respectively
- the input of the `SubByte` function in the first AES round as x_i, equal to $p_i \oplus k_i$, where p_i and k_i respectively represent the plaintext byte and the corresponding (first round) key byte, $0 \leq i \leq 15$

- the XOR addition with $\oplus$
- the multiplication over the Galois Field with 256 elements with $\bullet$
- $CT^{(i)}$ refers to the i-th S-box cache event (in the encryption of the plaintext P under the key K, implicitly stated). Its possible values are H and M for cache hit and miss respectively.

We represent an exclusive disjunction of statements with $\Bigg\langle$, meaning that *one and only one* of them is valid, whereas the conjunction of statements is expressed with $\Bigg\{$, where *all* statements are valid.

We index the bytes row-wise and *not* column-wise as in the AES specification [36, 42], i.e. in our notation, p_0, p_1, p_2, p_3 is the first row of a 16-byte plaintext (we assume that in an embedded software AES implementation S-box lookups are performed row-wise, so indexing bytes in the order of S-box computation simplifies the description of our algorithms).

2.2.5 The Advanced Encryption Standard

In 2000, the Rijndael algorithm [42] was chosen by the NIST to replace the widely deployed but ageing DES [41] and its more secure variant 3-DES as the block cipher of reference, after a long selection process based on resistance to standard cryptanalysis and efficiency of the design.

We outline below a few properties on the Rijndael structure that we utilize in this chapter. The block cipher has a fixed size state of 128 bits and comes with three possible key lengths: 128, 192 and 256 bits, the first one being the standard one. The number of rounds are respectively 10, 12 and 14. The 16 bytes block is treated as a 4 by 4 matrix. Each round is composed of four operations:

1. `AddRoundKey`: XORes the state with the round key (derived from the master key $K^{(0)}$ via the key schedule).
2. `SubBytes`: A non-linear byte substitution: composing an affine transformation with an inversion in the finite field with 256 elements. For performance reasons, this more complex operation is often implemented as a 256-byte lookup table.
3. `ShiftRows`: The four rows of the state matrix are shifted by 0, 1, 2 and 3 positions respectively.
4. `MixColumns`: A fixed matrix is multiplied with the columns of the state matrix modulo a fixed polynomial, whose bytes are again seen as elements of the finite field with 256 elements.

In the last round, the `MixColumns` operation is skipped. The pseudo-code of the AES-128 encryption routine is shown in Algorithm 1.

Algorithm 1 Encryption with Rijndael – AES-128

```
Input: Plaintext, Key
$r \in [0, 10]$, $K^{(r)} \leftarrow$ KeySchedule(Key)
State ← Plaintext
State ← AddRoundKey(State, $K^{(0)}$)
for $r$ from 1 to 9 do
    State ← SubBytes(State)
    State ← ShiftRows(State)
    State ← MixColumns(State)
    State ← AddRoundKey(State, $K^{(r)}$)
end for
State ← SubBytes(State)
State ← ShiftRows(State)
Ciphertext ← AddRoundKey(State, $K^{(10)}$)
Output: Ciphertext
```

The key schedule, also called key expansion, is a bijective map recursively applied to the master key to produce 10 more round keys $K^{(1)}, \ldots, K^{(10)}$ (the first round key $K^{(0)}$ being the master key). It is composed of the SubBytes function, some word rotation and XOR additions. A detailed description of the key schedule can be be found in the literature [36, 42, 76].

2.2.6 Our assumptions about the cache mechanism

Here we present our assumptions about the cache mechanism and the implementation of the AES lookup table that will be used in our attacks. In general, we follow the description made by Bonneau [23] and the assumptions of Fournier and Tunstall [45].

We assume that the AES implementation uses lookup tables of 256 entries. Let b be the size of a table entry in bytes. In case of a standard S-Box implementation, $b = 1$ (unless the 8-bit entries are stored as words of native length for the platform), in case of T-tables used in optimized implementations [36], $b = 4$. Let l be the cache line size in bytes. In modern embedded microcontrollers, common sizes are $l = 16$ and $l = 32$. Then, we have $\delta = l/b$ entries per cache line and $m = 256 \cdot b/l$ cache blocks per lookup table (note that $\delta m = 256$). The value of δ (or, equivalently, m) has effect on the attack complexity (described by Bonneau [23]) since cache events are determined by equations and inequations of $8 - \log_2 \delta$ higher order bits of the inputs to the lookups. We assume that the lookup table is aligned with the cache. As shown by Zhao and Wang, the attack is also possible when lookup tables are misaligned [123].

We present our attacks for the case $b = 1$, $l = 16$, which is a sensible configuration in the case of a constrained device. The attacks are adaptable to other cache and lookup table configurations. We carry out a theoretical analysis (Section 2.8) in general for different cache line sizes, i.e. for different values of m.

We assume that in an embedded software AES implementation S-Box lookups are performed row-wise (and therefore our row-wise notation simplifies the description of the attack algorithms). The adversary is dealing with a sequence of *observed* cache events, misses or hits, occurred in the first two rounds of the implementation. We call this sequence a *cache trace*. A cache trace can be recovered from a side-channel trace, as we show in Section 2.3 with EM measurements on a 32-bit microcontroller. Since there may be uncertainties in distinguishing a miss from a hit, we also introduce an additional type of observed cache event: the uncertain event.

Like the attacks of Bonneau [23], our attacks do not necessarily require the cache to be clean of lookup table entries prior to each run of the implementation, but can be made more efficient under the clean cache assumption.

2.3 Cache events in side-channel leakage

In this section, we describe the experiments we conducted on a microcontroller exploring the influence of the cache mechanism on electromagnetic (EM) measurements through a very simple setup.

The device we considered was an Olimex LPC-H2124 development board [80] (Figure 2.2b) carrying the NXP LPC2124 [79], an ARM7 microcontroller. Though ARM7 family devices do normally have a cache, this particular microcontroller features a Memory Accelerator Module (MAM). The MAM is a 128 bits wide cache that increases the latency of access to the onboard flash memory. In Figure 2.2a we present the EM traces acquired while the microcontroller with MAM enabled was performing a series of lookups in the AES S-box table that was stored in the flash memory. The acquisition was performed with Langer RF-B 0.3-3 H-field probe, Langer PA 203 20 dB pre-amplifier and LeCroy WaveMaster 104MXi oscilloscope. The CPU clock frequency of the microcontroller was 59 MHz, the sampling rate of the Digital Sampling Oscilloscope (DSO) was set to 5 GS/s (5 billion samples per second). The probe was fixed by a lab stand with the probe tip touching the surface of the microcontroller package; the precise position of the probe was determined experimentally.

The top trace shows a sequence of 3 cache misses, whereas the bottom trace shows a miss-hit-miss sequence. Cache misses can be seen as distinguished peaks. Note also the timing differences: Figure 2.2a suggests that the cache hit takes 2 CPU clock cycles less than the cache miss. We would like to stress that the traces were acquired *without averaging* in an *unshielded* setup depicted in Figure 2.2b.

Since the MAM is a single-line cache, the attacks we propose in this chapter cannot be implemented with this microcontroller, as opposed to the original chosen plaintext attack [45]. Still, our experiments are a sound example of cache event leakage for an ARM microcontroller, which has not been considered in earlier works except by Rebeiro and Mukhopadhyay [91], exploiting cache event leakage in the power consumption of a PowerPC processor within a Xilinx Virtex-II FPGA to mount an attack against CLEFIA.

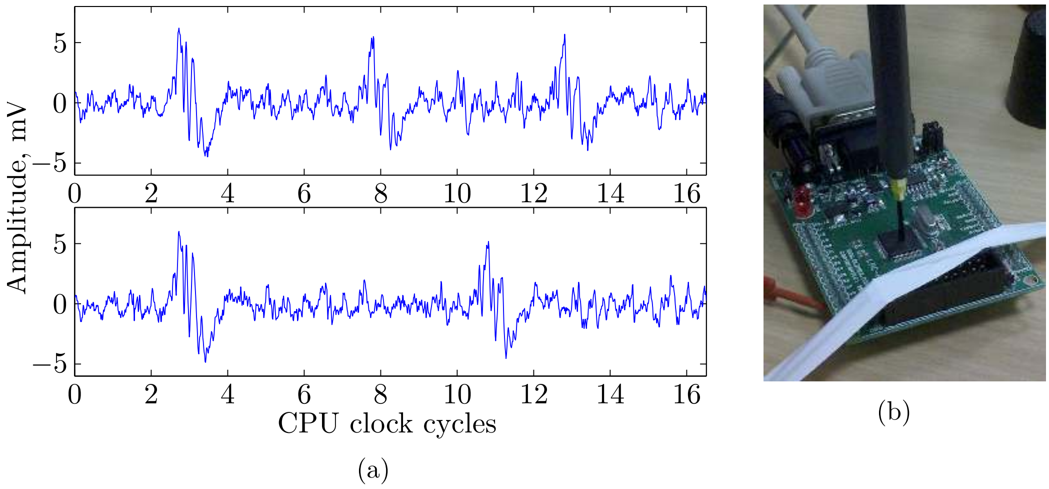


Figure 2.2: (a) EM traces of an ARM7 microcontroller with distinguishable cache event sequences: miss-miss-miss (top) versus miss-hit-miss (bottom); (b) the μC with the passive EM probe

2.4 Chosen plaintext attacks

In this section, we present two attacks where the adversary is able to submit plaintexts of his choice to an AES encryption running on the target device. The plaintexts are adaptively chosen after each encryption depending on the previous plaintexts and obtained cache sequences. The first attack presented in this section is from Tunstall and Fournier [45]. The second attack is our improved attack which takes significantly less measurements than the original one. We finally explain the limitations of these adaptive strategies. Throughout this section, the attacks are described assuming error-free measurements and a clean cache.

2.4.1 Adaptive chosen plaintext attack of ACISP'06

From the description of the AES (Section 2.2.5) and our assumptions (Section 2.2.6), we observe that the first lookup to the S-box table occurs in the first AES round and is indexed by the value $p_0 \oplus k_0$. Because the cache is clean, we know that a cache hit (H) will occur at the second lookup $CT^{(1)}$ only if the upper nibble of $p_1 \oplus k_1$ equals that of $p_0 \oplus k_0$. Using our notations, $CT^{(1)} = H$ implies

$$\widehat{p_1 \oplus k_1} = \widehat{p_0 \oplus k_0}$$

We can rearrange the terms in the latter equality to obtain

$$\widehat{k_0 \oplus k_1} = \widehat{p_0 \oplus p_1}$$

The high nibble $\widehat{p_1}$ that generates a cache hit at the second lookup can be quickly identified because there exists only one value out of 16 which can generate a cache hit (as only one line is loaded in the cache). This high nibble will come up after an average number of $\sum_{i=1}^{16} \frac{i}{16} = 8.5$ acquisitions. Once the desired $\widehat{p_1}$ is found, the adversary can reiterate the process for the third and subsequent lookups, generating a cache sequence being $MH \dots H$. At the end of this first round stage, she then knows the values $(\widehat{p_i})_{1 \leq i \leq 15}$ such that:

$$\forall i \in [0, 15],\ \widehat{k_0 \oplus k_i} = \widehat{p_0 \oplus p_i}$$

The steps of the attack are detailed in Algorithm 2.

Algorithm 2 Chosen plaintext attack from ACISP'06 [45]

```
$\boldsymbol{P} = (p_0, p_1, \dots, p_{15}) \leftarrow (0, 0, 0, \dots, 0)$          ▷ Plaintext
$i \leftarrow 1$
while $i < 16$ do
    $\boldsymbol{CT} \leftarrow \mathrm{AES}_k(\boldsymbol{P})$          ▷ Acquisition
    if $CT^{(i)} = H$ then
        $i$++
    else
        $\widehat{p_i}$++
    end if
end while
Output: $\boldsymbol{P}$
```

We can express $\widehat{p_i}$ as a function of $\widehat{p_0}$:

$$\widehat{p_i} = \widehat{p_0} \oplus \widehat{k_i} \oplus \widehat{k_0}$$

Thus for clarity in the description, Algorithm 2 arbitrary begins with the zero plaintext.

Following this strategy an adversary deduces $15 \times 4 = 60$ bits of information in the first round, reducing the entropy of the key from 128 to 68. The best, average and worst number of plaintexts (traces) required are respectively 1, $15 \times 8.5 = 127.5$ and $16 \times 15 = 240$. The distribution of the number of inputs is plotted in Figure 2.3, obtained from a simulation carried out over 10^5 random keys. We show below how to improve these figures.

2.4.2 Our improvement

Recall that in a chosen plaintext approach, the adversary aims at finding a plaintext block that generates a cache miss immediately followed by a series of cache hits. In other words, all lookups performed in the first round must correspond to the same cache line. In the attack from ACISP'06, only one lookup is exploited for each query, although more information is contained in the subsequent lookups. Amongst them, a second cache miss indeed refers to another loaded line in the cache, thus the corresponding plaintext nibble should be updated because only one loaded cache line is desired. On the other hand, a cache hit *may* or *may not* refer to the first loaded line, thus the adversary cannot draw any information from a cache hit.

We now explain how to build up constraints on the plaintext bytes from all cache misses so as to come up with the desired plaintext with only a few acquisitions.

For two given distinct positions i and j where cache misses (denoted by M) occur, we have $\widehat{p_i \oplus k_i} \neq \widehat{p_j \oplus k_j}$. As a result, the adversary must set aside all plaintexts with the particular difference $\delta = \widehat{p_i \oplus p_j}$ as such plaintexts induce a hit. That is, she can evict the value δ (out of 16 initially) for the upper nibble of the XOR difference between p_i and p_j. Proceeding, a set of constraints on the plaintext nibbles can be built.

The steps of our improved attack are detailed in Algorithm 3. The constraints on every pair of plaintext nibbles i and j, $j > i$, are represented with a triangular matrix $\boldsymbol{\Delta}$ that we fill with the values to avoid for the upper nibble of one XOR difference $\widehat{p_i \oplus p_j}$. The subroutine `SelectNextPlaintext` updates the plaintext $\boldsymbol{P}$ according to the constraints $\boldsymbol{\Delta}$. Namely, it updates the nibble values verifying that the value $\delta = \widehat{p_i \oplus p_j}$ is not forbidden, thus going much faster than in the attack from ACISP'06. As one can see, Algorithm 3 resembles Algorithm 2 with an additional `for` loop at lines 7–15 that builds up the constraints on $\widehat{p_i \oplus p_j}$.

Algorithm 3 Improved chosen plaintext trace-driven cache-collision attack

```
 1: $\boldsymbol{P} = (p_0, p_1, \ldots, p_{15}) \leftarrow (0, 0, \ldots, 0)$          ▷ Plaintext
 2: $\boldsymbol{\Delta} = \{\Delta_{i,j}\}_{\substack{1 \leq i \leq 15 \\ 1 \leq j \leq i}} \leftarrow \{\emptyset\}_{\substack{1 \leq i \leq 15 \\ 1 \leq j \leq i}}$          ▷ Constraints
 3: $i \leftarrow 1$
 4: while $i < 16$ do
 5:     $\boldsymbol{P} \leftarrow$ SelectNextPlaintext$(i, \boldsymbol{P}, \boldsymbol{\Delta})$
 6:     $\boldsymbol{CT} \leftarrow \mathrm{AES}_k(\boldsymbol{P})$          ▷ Acquisition
 7:     for $j$ from $i$ to 15 do
 8:         if $CT^{(j)} = M$ then
 9:             for $l$ from 0 to $j - 1$ do
10:                 if $CT^{(l)} = M$ then
11:                     $\Delta_{j,l} \leftarrow \Delta_{j,l} \cup \widehat{p_j \oplus p_l}$
12:                 end if
13:             end for
14:         end if
15:     end for
16:     while $(CT^{(i)} = H) \wedge (i < 16)$ do
17:         $i$++
18:     end while
19: end while
Output: $\boldsymbol{P}$
```

We have simulated this attack with 10^5 random keys. From the original attack [45] to our improved method, the average number of required inputs to obtain a 60-bit reduction of the key entropy significantly decreases: from 127.5 to 14.5.

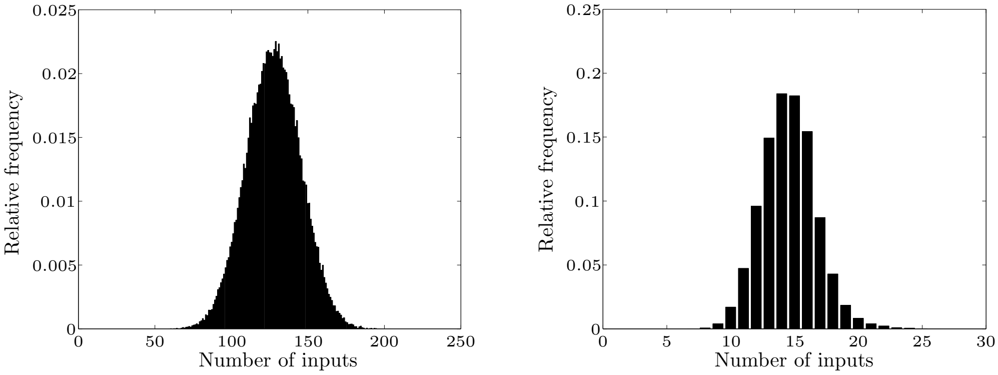


Figure 2.3: Distribution of the number of plaintexts required to obtain a 60-bit reduction of the key search space

2.4.3 Drawback of the chosen plaintext strategy

The main disadvantage of a chosen plaintext attack resides in its high complexity to exploit the second round lookups.

To recover the remaining 68 bits of the key, the analysis of the second round lookups also follows a chosen plaintext strategy, continuing to look for plaintexts leading to cache hits in the first lookups of the second round, as proposed by Fournier and Tunstall [45]. In the second round, $\widehat{k_0}$ and the low key nibbles are involved in the lookup addresses (their detailed expressions are given in Section 2.5.2), hence resolving these equations further reduces the key entropy.

Nevertheless, a high number of measurements is required to sufficiently reduce the search space. At the first lookup of the second round, 6 unknown nibbles are involved, which means that 6 equations are needed to find the correct combination of nibbles. A cache hit occurs with probability 1/16, thus 96 measurements are required on average. The key entropy is then reduced to $68 - 24 = 44$.

At the second lookup, the 5 unknown nibbles can be recovered in the same manner, but a consecutive cache hit will only occur with probability 1/256, which means that the average number of acquisitions required is now 1280. Thereafter, an exhaustive search over 2^{24} key hypotheses remain.

Unlike the first round stage, this cannot be improved: one cannot introduce an efficient way of inducing constraints on the plaintexts due to the non-linearity of the equations emerging from the second round lookups. Hence, our improvement on the first round has an insignificant effect on the overall complexity of the original plaintext attack. Therefore, we had to come up with a known plaintext attack which we describe in the next section.

2.5 Known plaintext attack

Studying the attack of ACISP'06 and our proposed improvement, several questions arise:

- How can an adversary gain information from both cache hits and misses?
- How can the second round lookups be better exploited?
- Can trace-driven cache attacks be applied to a known plaintext scenario?

As an answer to these questions, we put forward a known plaintext attack that exploits all cache events including hits and the four first second round lookups. We make the same implementation assumptions on the table width (16 bytes) and on the S-box size (256 bytes).

Two distinct phases, first online, second offline, compose the attack. First, N random plaintexts[1] are submitted to the target device for an AES encryption under the unknown key and the side-channel traces are converted to cache traces. Second, the cache events of the first round and the first four lookups of the second round are analysed.

Throughout this section, we assume that: (1) the cache is clean prior to encryption, which can be ensured by resetting the device; (2) no mistake is made in the conversion of side-channel traces to cache traces. We later release these constraints in Section 2.6.

2.5.1 Analysis of the first round

Our analysis of the first round can be viewed as a sieve, i.e. an algorithm that gradually reduces for each variable the number of possible candidates to one, as long as enough information is provided in the acquisitions. The sieve takes as input the N plaintexts (denoted $\boldsymbol{P}^{(q)} = (p_i)^{(q)}_{0 \leq i \leq 15})$ that served in the online phase for the encryptions under the unknown key $K = (k_0, k_1, \ldots, k_{15})$ and the N cache traces converted from the side-channel measurements (denoted $\boldsymbol{CT}^{(q)} = (CT_i)^{(q)}_{0 \leq i \leq 15})$. The sieve outputs a set of linear equations in the high nibbles of k_i $(0 \leq i \leq 15)$ that decreases the entropy of the key search space by $15 \times 4 = 60$ bits.

We recall that a cache *miss* allows an adversary to know that the accessed line is different from all the lines previously accessed in the encryption. Because in our framework the table lines are indexed by the high nibble of an S-box input, a cache miss at the i-th lookup is algebraically expressed by the following inequations:

$$\forall j \in \Gamma,\ \widehat{k_i \oplus p_i} \neq \widehat{k_j \oplus p_j} \tag{2.1}$$

where Γ denotes the set of indexes where a cache miss occurred previously in the encryption.

Analogically, a cache *hit* allows an adversary to know that the accessed line belongs to one of the lines previously accessed in the encryption. Therefore, a cache hit at the i-th lookup induces the following equations:

$$\exists! j \in \Gamma,\ \widehat{k_i \oplus p_i} = \widehat{k_j \oplus p_j} \tag{2.2}$$

[1] The attack will work in the same manner against an AES decryption, with the submission of random ciphertexts.

The two statements (2.1) and (2.2) can also be written as follows:

$$CT_i = M \implies \forall j \in \Gamma,\ \widehat{k_i \oplus k_j} \neq \widehat{p_i \oplus p_j} \tag{2.3}$$

and

$$CT_i = H \implies \exists! j \in \Gamma,\ \widehat{k_i \oplus k_j} = \widehat{p_i \oplus p_j} \tag{2.4}$$

Furthermore, we have the following triangular relation:

$$\widehat{k_i \oplus k_j} = \widehat{k_i \oplus k_0} \oplus \widehat{k_j \oplus k_0} \tag{2.5}$$

We inject (2.5) in the statements (2.3) and (2.4) so as to finally obtain:

$$CT_i = M \implies \forall j \in \Gamma,\ \widehat{k_i \oplus k_0} \neq \widehat{p_i \oplus p_j} \oplus \widehat{k_j \oplus k_0} \tag{2.6}$$

and

$$CT_i = H \implies \exists! j \in \Gamma,\ \widehat{k_i \oplus k_0} = \widehat{p_i \oplus p_j} \oplus \widehat{k_j \oplus k_0} \tag{2.7}$$

On the right-hand sides of statements (2.6) and (2.7), $\widehat{p_i \oplus p_j}$ is known and if $\widehat{k_j \oplus k_0}$ were known too, information on $\widehat{k_i \oplus k_0}$ would be easily deduced.

This suggests to process the cache traces position-wise, instead of acquisition-wise as in the described chosen plaintext attacks. That is, starting from position $i = 1$, an adversary should use the statements (2.6) and (2.7) along with as many cache traces as necessary to retrieve the correct value of $\widehat{k_i \oplus k_0}$. Only then, she can carry on to the next position and repeat the process until $i = 15$.

The advantage of this strategy for an adversary is the capability, once the set of possibilities for a XOR difference has been reduced to a singleton, to reuse this information while analysing the successive lookups. Compared to the adaptive strategies, the complexity of the attack is drastically reduced in terms of memory and time because we aim at retrieving the value of $\widehat{k_i \oplus k_0}$ only once $\widehat{k_j \oplus k_0}$ are known for every $j < i$. Here, every atom of information on $\widehat{k_j \oplus k_i}$ directly reduces the set of candidates for $\widehat{k_0 \oplus k_i}$.

Our sieve is formally detailed in Algorithm 4. The set $\boldsymbol{\kappa}_i$ denotes the set of possible values for $\widehat{k_i \oplus k_0}$, initiated to $\{0, \ldots, 15\}$.

Finally, the high nibbles of the key bytes form a system of 15 linearly independent equations, reducing the entropy of the key down to $128 - 4 \times 15 = 68$ bits.

We estimated the required number of acquisitions out of 10^5 simulated attacks each with a random key. Our results are illustrated in Figure 2.4. For each experiment, the number of required inputs is the maximum number of inputs required for recovering the nibble differences $\widehat{k_0 \oplus k_i}$. The average number of inputs allowing a 68-bit recovery was 19.43, which is notably less than the 127.5 inputs of the original chosen plaintext attack [45] described in Section 2.4.1.

Interestingly, our method also works if cache hits are not taken into account in the analysis, that is, if the set $\boldsymbol{\kappa}_i$ remains unchanged after the observation of a cache hit at position i. The execution of the sieve then requires more inputs, namely 54.19 to achieve the 60-bit reduction. However, it was shown by Bonneau [23] that using only misses allows an adversary to conduct a trace-driven cache attack even if the cache already contains S-box lookup table lines prior to the encryption. We will elaborate on this in Section 2.6.4.

Algorithm 4 Known plaintext analysis of the first round

```
Input: $(\boldsymbol{P}^{(q)}, \boldsymbol{CT}^{(q)}) = (p_i^{(q)}, CT_i^{(q)})_{0 \le i \le 15},\ q \in [1, N]$
$\boldsymbol{\kappa}_i \leftarrow \{0, \ldots, 15\},\ 1 \le i \le 15$
for $i \leftarrow 1$ to 15 do
    $q \leftarrow 0$
    while $|\boldsymbol{\kappa}_i| > 1$ do
        $q \leftarrow q + 1$
        $\boldsymbol{\kappa}' \leftarrow \emptyset$
        for $j \leftarrow 0$ to $i - 1$ do
            if $CT_j^{(q)} = M$ then
                $\boldsymbol{\kappa}' \leftarrow \boldsymbol{\kappa}' \cup \left\{ \widehat{p_i^{(q)} \oplus p_j^{(q)}} \oplus \boldsymbol{\kappa}_j \right\}$
            end if
        end for
        if $CT_i^{(q)} = M$ then
            $\boldsymbol{\kappa}_i \leftarrow \boldsymbol{\kappa}_i \setminus \boldsymbol{\kappa}'$
        else
            $\boldsymbol{\kappa}_i \leftarrow \boldsymbol{\kappa}_i \cap \boldsymbol{\kappa}'$
        end if
    end while
end for
Output: $\boldsymbol{\kappa}_i,\ i \in [1, 15]$
```

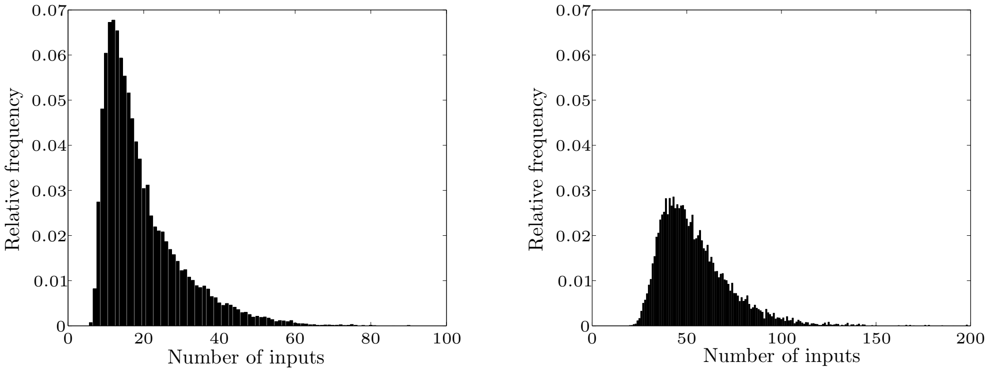
2.5.2 Analysis of the second round

In this section, we show that the first four lookups of the second round can also be exploited in a known plaintext scenario and the key entropy further reduced. Our approach is a natural extension to the one performed in the first round, but finding the correct lookup indexes requires significantly more computational effort. Our approach was briefly sketched by Acıiçmez and Koç [1], but they did not present the analysis of the number of acquisitions required, whereas we perform a theoretical analysis in Section 2.8. In the following description, we assume that the round keys are pre-computed and pre-stored, thus no access to the S-box lookup table is performed between the encryption rounds[2].

The second round analysis reuses the plaintexts and cache traces from the first round analysis, but in most cases more inputs are required. The five steps of this second stage are the analysis of the first four lookups and possibly finish with a little exhaustive search:

1. from the first lookup, recover $\widehat{k}_0$, $\check{k}_0$, $\check{k}_5$, $\check{k}_7$, $\check{k}_{10}$, $\check{k}_{15}$, 24 bits in total;

[2] Note that the strategy we propose here would be straightforward to adapt to an AES implementation with an on-the-fly key schedule. Similarly, the `xtimes` operation in the AES `MixColumn` can also be taken into account in case the former is implemented as a lookup table (Fournier and Tunstall also presented a chosen plaintext attack exploiting the `xtimes` lookup table).



(a) (b)

Figure 2.4: Distribution of the number of plaintexts required to obtain a 60-bit reduction of the key space in the known plaintext attack, considering hits and misses (a) and misses only (b)

2. from the second lookup, recover $\check{k}_1$, $\check{k}_6$, $\check{k}_{11}$, $\check{k}_{12}$, 16 bits in total;
3. from the third lookup, recover $(\check{k}_2, \check{k}_8, \check{k}_{13})$, 12 bits in total;
4. from the fourth lookup, recover $(\check{k}_3, \check{k}_4, \check{k}_9, \check{k}_{14})$, 16 bits in total;
5. at this point, between 1 and 10 entire key candidates remain. The correct one may have to be identified using a known plaintext-ciphertext pair.

Indeed, it can happen that several full key candidates pass the steps 1–4, although all the key nibbles are involved in the corresponding equations. This is due to the properties of the AES S-box: for some input high nibble, there can exist up to 4 output values which have the same high nibble. Hence, these values are indistinguishable in our equations because only the high nibble of an S-box input determines whether the cache event is a miss or a hit. We experimentally observed that the "finalists" that reach the fifth stage of the second round are at the number of 1 to 10.

We clarify the five steps of the second round analysis.

First lookup of the second round

The first lookup is indexed by:

$$y_0 = 2 \bullet s(x_0) \oplus 3 \bullet s(x_5) \oplus s(x_{10}) \oplus s(x_{15}) \oplus s(k_7) \oplus k_0 \oplus 1.$$

Therefore, if the first lookup of the second round is a miss, i.e. the cache is of the form $M * * \ldots * |M$, the following system of inequations holds:

$$\left\{ \begin{array}{ccc} \widehat{y}_0 & \neq & \widehat{x}_{j_1} \\ & \vdots & \\ \widehat{y}_0 & \neq & \widehat{x}_{j_L} \end{array} \right. , \quad j_1, \ldots, j_L \in \Gamma,$$

where Γ is the set of indices of misses observed in the 16 lookups of the first round, $L = |\Gamma|$. We can rearrange the inequations as below:

$$2 \bullet s(x_0) \oplus 3 \bullet s(x_5) \widehat{\oplus s(x}_{10}) \oplus s(x_{15}) \oplus s(k_7) \neq \left\{ \begin{array}{c} \widehat{\alpha}_{j_1} \\ \vdots \\ \widehat{\alpha}_{j_L} \end{array} \right. , \quad j_1, \ldots, j_L \in \Gamma, \tag{2.8}$$

where $\widehat{\alpha}_j$ are known values which depend on the plaintext bytes and the XOR differences of key nibbles recovered in the first part of the analysis. Similarly, a case this first lookup is a cache hit, i.e. for a cache trace of the form $M ** \ldots * |H$, we have:

$$\left\langle \begin{array}{ccc} \widehat{y}_0 & = & \widehat{x}_{j_1} \\ & \vdots & \\ \widehat{y}_0 & = & \widehat{x}_{j_L} \end{array} \right. , \quad j_1, \ldots, j_L \in \Gamma,$$

which becomes after rearrangement:

$$2 \bullet s(x_0) \oplus 3 \bullet s(x_5) \widehat{\oplus s(x}_{10}) \oplus s(x_{15}) \oplus s(k_7) = \left\langle \begin{array}{c} \widehat{\alpha}_{j_1} \\ \vdots \\ \widehat{\alpha}_{j_L} \end{array} \right. , \quad j_1, \ldots, j_L \in \Gamma. \tag{2.9}$$

In the left part of (2.8) and (2.9), the nibbles $\widehat{k}_0$, $\check{k}_0$, $\check{k}_5$, $\check{k}_7$, $\check{k}_{10}$, $\check{k}_{15}$ are unknown, for a total of 24 unknown bits. Solving (2.8) or (2.9) for a single trace by exhaustive search over the 2^{24} combinations of these nibbles leaves us with a fraction of possibilities. Another trace provides another system of (in)equations on the unknown nibbles and further reduces the amount of candidates.

After several traces, we will remain with the key bytes k_0, k_5, k_{10} and k_{15} fully recovered. However, as explained before, the AES S-box is such that depending on the input high nibble, 1, 2, 3 or 4 output values may have the same high nibble. Therefore, when the S-box input is only indexed by the key and not mixed with a plaintext byte, as for k_7 in (2.8) and (2.9), an adversary cannot distinguish the correct candidate for $\check{k}_7$ among these up to 4 possibilities and has to take into account this multiplicity of candidates for $\check{k}_7$ through the rest of the analysis. Only an exhaustive search conducted over full key candidates will determine the correct value of $\check{k}_7$.

2.5.3 Second lookup of the second round

Once done with the analysis of the first lookup, we proceed with the analysis of the second lookup. This lookup is indexed by:

$$y_1 = 2 \bullet s(x_1) \oplus 3 \bullet s(x_6) \oplus s(x_{11}) \oplus s(x_{12}) \oplus s(k_7) \oplus k_0 \oplus k_1 \oplus 1$$

Similarly, the nature of the corresponding cache event either leads to a system of inequations if it is a cache miss (after rearranging the terms):

$$2 \bullet s(x_1) \oplus 3 \bullet \widehat{s(x_6) \oplus} s(x_{11}) \oplus s(x_{12}) \neq \left\{ \begin{array}{c} \widehat{\alpha}_{j_1} \\ \vdots \\ \widehat{\alpha}_{j_R} \end{array} \right. , \quad j_1, \ldots, j_R \in \Gamma, \tag{2.10}$$

either to a system of equations if the lookup is a cache hit:

$$2 \bullet s(x_1) \oplus 3 \bullet \widehat{s(x_6) \oplus} s(x_{11}) \oplus s(x_{12}) = \left\langle \begin{array}{c} \widehat{\alpha}_{j_1} \\ \vdots \\ \widehat{\alpha}_{j_R} \end{array} \right. , \quad j_1, \ldots, j_R \in \Gamma, \tag{2.11}$$

where Γ is the set of indexes of misses observed in the 17 previous lookups (i.e. in the first round and in the first lookup of the second round), with $R = |\Gamma|$ and $\widehat{\alpha}_j$ are some known values depending on the plaintext bytes and the previously recovered nibbles of the key bytes. The only difference with the analysis of the first lookup arises in case the first lookup of the second round is a miss, where the (in)equations in (2.10) or (2.11) include either $\widehat{y_1} \neq \widehat{y_0}$ or $\widehat{y_1} = \widehat{y_0}$. However, the value of y_0 is known from the analysis of the first lookup, thus this (in)equation can be considered.

We have only 16 unknown bits in (2.10) and (2.11), namely in the nibbles $\check{k}_1$, $\check{k}_6$, $\check{k}_{11}$ and $\check{k}_{12}$, the rest having been recovered in the previous steps. By solving the equations for several traces like in the analysis of the first lookup, we will get a single candidate for these unknown nibbles, notwithstanding the multiplicity of candidates coming from k_7 in the first lookup. We note that this multiplicity is not further increased here because the S-box inputs all include a plaintext byte.

After the analysis of the first two lookups of the second round, the key chunks $\check{k}_2$, $\check{k}_3$, $\check{k}_4$, $\check{k}_8$, $\check{k}_9$, $\check{k}_{13}$ and $\check{k}_{14}$ remain unknown. They comprise 28 bits. Considering up to 4 possible values for k_7, the key space is now reduced to 2^{30} elements. An exhaustive search can already be launched. However, analysing the two subsequent lookups has a lower computational complexity and reveals all the unknown bits, except for $\check{k}_7$.

2.5.4 Third and fourth lookups of the second round

The third lookup is indexed by:

$$y_2 = 2 \bullet s(x_2) \oplus 3 \bullet s(x_7) \oplus s(x_8) \oplus s(x_{13}) \oplus s(k_7) \oplus k_0 \oplus k_1 \oplus k_2 \oplus 1$$

where the nibbles $\check{k}_2$, $\check{k}_8$ and $\check{k}_{13}$, thus 12 bits, are unknown. Like for the first two lookups, an adversary can run through the possible combinations of these nibbles and, after several traces, retain the only candidate that satisfies the systems of (in)equations. Again, $\widehat{y_0}$ and $\widehat{y_1}$ are possibly involved but known from the previous steps.

The fourth lookup is indexed by:

$$y_3 = 2 \bullet s(x_3) \oplus 3 \bullet s(x_4) \oplus s(x_9) \oplus s(x_{14}) \oplus s(k_7) \oplus k_0 \oplus k_1 \oplus k_2 \oplus k_3 \oplus 1$$

where the unknown nibbles $(\check{k}_3, \check{k}_4, \check{k}_9, \check{k}_{14})$, 16 bits in total, are determined in the same manner.

Finally, the correct key is recovered with a search among up to 10 key candidates, requiring the knowledge of a valid plaintext-ciphertext pair.

Like the first round analysis, the second round analysis can exploit only the cache misses so as to be robust to the pre-loaded cache condition.

2.5.5 Attack complexity

The complexity of the first round analysis is negligible. As we already described in Section 2.5.1, it takes on average 19 measurements and the solving of the sieve is instantaneous on a PC.

In the second round analysis, we have to evaluate expressions like in (2.8) $O(2^{24})$ times for each required trace, times the multiplicity of k_7. The required number of traces has been estimated through simulations; it is about 29 traces (reusing the traces from the first round part) in case both hits and misses are used. This part runs in less than a minute on a PC.

The final exhaustive search has a negligible complexity.

We will present a more detailed estimation of the attack complexity after introducing our adaptations to real-life conditions in the next section. We will also present the theoretical model for determining the measurement complexity (Section 2.8), which is confirmed by our simulations.

2.6 Error tolerance

In this section, we elaborate on the adaptations of our attacks in order to make them resistant to errors. We begin with a description of our approach to deal with detection uncertainties and we present the modifications brought to our attacks. Then we explain how to treat cache information generated with partially pre-loaded cache and finally we present the results of our simulations considering different levels of these real-life complications.

2.6.1 General approach to distinguishing cache events

The generation of a cache sequence from a power trace is a task that eventually needs to be automated in case of treating a high number of acquisitions. Moreover, although our experiments have exhibited clear differences of amplitude and timing from the two kinds of cache events and an obvious ability for an attacker to distinguish between them on an EM trace, the cache events may happen to be more tricky to distinguish on other platforms and measurement setups.

For these two reasons, we introduce a treatment of the side-channel traces which uses a statistic (e.g. the absolute value of the trace, if necessary averaged over a defined interval) to convert them to cache sequences with a tolerance to noise. According to our experiments, we can reasonably assume this statistic to be smaller in case of a cache hit and larger in case of a cache miss. Moreover, as the noise is generally Gaussian, cache hits and misses are also expected to follow a Gaussian distribution. Hence, the task for the adversary is to distinguish between two normal distributions.

Setting a single threshold to dissociate them will unavoidably induce the detection of a hit for a miss and vice-versa, also called in statistics Type I and Type II errors. However, our algorithms do not hold in presence of these errors as the latter would introduce false equations and inequations. Hence it is necessary to define two thresholds in the detection and a resulting third kind of event, called "uncertain". In the rest of the section, we describe how to adapt our algorithms so that they also deal with the "uncertain" event, determined by the two thresholds t_H and t_M as follows:

1. If the statistic is smaller than t_H, the event is considered as a hit.
2. If the statistic is greater than t_M, the event is considered as a miss.
3. If the statistic falls between t_H and t_M, the event is considered as uncertain.

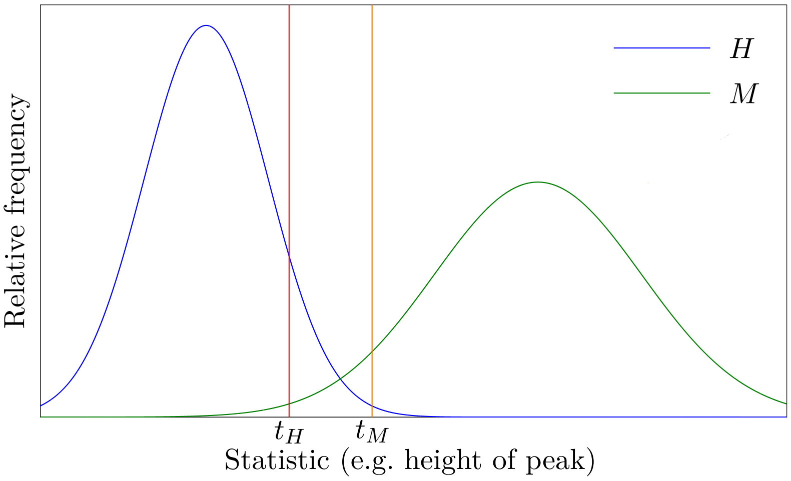


Figure 2.5: Probability density functions of cache hits (H) and misses (M)

t_H and t_M are chosen so that it is very unlikely that a miss is interpreted as a miss and the other way around. In Figure 2.5, typical probability density functions for the statistic of power traces of a cache hit (H) or a cache miss (M) are represented along with the thresholds t_H and t_M. Obviously, the greater the probability that the statistic of a cache event falls between these thresholds, the more acquisitions are required to retrieve the key, as we verify in our simulations in Section 2.6.5.

2.6.2 Error-tolerant chosen plaintext attack

The improved chosen plaintext attack (Algorithm 3) naturally adapts itself to the presence of uncertain events. For the cache event being analysed, the occurrence of an uncertain event forces an attacker to drop the analysis of the trace and repeat the acquisition. In the analysis of the subsequent positions, cache misses can nevertheless be taken into account in the constraints, whereas uncertain events occurring at these positions have to be treated as *hits* so that no false equation is added to the set of constraints.

To observe the influence of this third type of cache event, we simulated the error-tolerant attack for random 10^4 keys. The results show that our chosen plaintext attack stands well the presence of error. On average, 22.6 traces were required when the occurrence frequency of an uncertain event was 0.2. When this frequency was 0.5, an average number of 47.2 traces was required. Note that this measurement complexity is lower than for the original adaptive algorithm without error resilience given by Fournier and Tunstall [45].

2.6.3 Error-tolerant known plaintext attack

We show below how to adapt the first and second stages of our known plaintext attack to take into account uncertain events along with cache misses and hits.

Adaptation of the first round

Analysing the lookup at position i, three cases may now occur:

1. CT_i is a cache **miss**. An uncertain event occurring in the same acquisition before position i should not be taken as misses, as they might lead to a false inequation on $\widehat{k_0 \oplus k_i}$, evicting the correct value out of $\boldsymbol{\kappa}_i$. Therefore, uncertain events occurring before position i have to be treated as hits. Otherwise stated, $\boldsymbol{\kappa}_i$ is updated to $\boldsymbol{\kappa}_i \setminus \{\widehat{k_0 \oplus k_j} \mid j \in \Gamma\}$, where Γ refers to the set of indices where previously occurred a cache miss.
2. CT_i is a cache **hit**. An uncertain event occurring in the same acquisition before position i should be treated as a miss, otherwise a false equation on $\widehat{k_0 \oplus k_i}$ might reduce $\boldsymbol{\kappa}_i$ to a set excluding the correct value for $\widehat{k_0 \oplus k_i}$. Therefore, uncertain events occurring before position i have to be treated as misses. Otherwise stated, $\boldsymbol{\kappa}_i$ is updated to $\boldsymbol{\kappa}_i \cap \{\widehat{k_0 \oplus k_j} \mid j \in \Gamma \cup \Upsilon\}$, where Υ refers to the set of indices where previously occurred an uncertain event.
3. CT_i is an **uncertain** event. No action can be taken, $\boldsymbol{\kappa}_i$ is left as is.

This strategy for the first round is formally expressed in Algorithm 5.

Adaptation of the second round

The second round adapts itself in the same manner. Here we denote Γ_r and Υ_r as the sets of indexes where previously occurred respectively a cache miss and an uncertain event, in round $r = 1, 2$. When analysing the lookup at position $i \in [0, 3]$ in the second round, the adversary faces three possibilities:

1. CT_i is a cache **miss**. For every possible combination of unknown nibbles involved in the computation of y_i:
$$\widehat{y_i} \notin \{\widehat{k_0} \oplus \widehat{k_0 \oplus k_j} \oplus \widehat{p_j} | j \in \Gamma_1\} \cup \{\widehat{y_j} | j \in \Gamma_2\}$$
2. CT_i is a cache **hit**. For every possible combination of unknown nibbles involved in the computation of y_i:
$$\widehat{y_i} \in \{\widehat{k_0} \oplus \widehat{k_0 \oplus k_j} \oplus \widehat{p_j} | j \in \Gamma_1 \cup \Upsilon_1\} \cup \{\widehat{y_j} | j \in \Gamma_2 \cup \Upsilon_2\}$$

Algorithm 5 Known plaintext analysis of the first round with uncertain cache events

```
Input: $(\boldsymbol{P}^{(q)}, \boldsymbol{CT}^{(q)}) = (p_i^{(q)}, CT_i^{(q)})_{0 \leq i \leq 15},\ q \in [1, N]$
$\boldsymbol{\kappa}_i \leftarrow \{0, \ldots, 15\},\ 1 \leq i \leq 15$
for $i \leftarrow 1$ to 15 do
    $q \leftarrow 0$
    while $|\boldsymbol{\kappa}_i| > 1$ do
        $q \leftarrow q + 1$
        $\boldsymbol{\kappa}', \boldsymbol{\kappa}^* \leftarrow \emptyset$
        for $j \leftarrow 0$ to $i - 1$ do
            if $CT_j^{(q)} = M$ then
                $\boldsymbol{\kappa}' \leftarrow \boldsymbol{\kappa}' \cup \left\{\widehat{p_i^{(q)} \oplus p_j^{(q)}} \oplus \boldsymbol{\kappa}_j\right\}$
            else if $CT_j^{(q)} = U$ then
                $\boldsymbol{\kappa}^* \leftarrow \boldsymbol{\kappa}^* \cup \left\{\widehat{p_i^{(q)} \oplus p_j^{(q)}} \oplus \boldsymbol{\kappa}_j\right\}$
            end if
        end for
        if $CT_i^{(q)} = M$ then
            $\boldsymbol{\kappa}_i \leftarrow \boldsymbol{\kappa}_i \setminus \boldsymbol{\kappa}'$
        else if $CT_i^{(q)} = H$ then
            $\boldsymbol{\kappa}_i \leftarrow \boldsymbol{\kappa}_i \cap (\boldsymbol{\kappa}' \cup \boldsymbol{\kappa}^*)$
        end if
    end while
end for
Output: $\boldsymbol{\kappa}_i,\ i \in [1, 15]$
```

3. CT_i is an **uncertain** event. No action can be taken to reduce the number of combinations of unknown nibbles leading to y_i.

The results of our simulations of the full attack are presented in Section 2.6.5.

2.6.4 Partially preloaded cache

Prior to an AES encryption, it may happen that some S-box elements are still present in the cache from a previous execution of the routine. Then, although a cache miss at position j still indicates that the values $k_j \oplus p_j$ or y_j—in the first and second round respectively—were not already present in the cache, a cache hit may refer to a line that was loaded during a previous AES encryption. Therefore, in case of a cache hit, no equation can be drawn as the correct hypothesis may be excluded from the set of possibilities. For this reason, an analysis solely exploiting the cache misses has to be adopted when the cache is partially preloaded. This strategy was also adopted by Bonneau [23].

The chosen plaintext attack (Section 2.4.2) is naturally suited for this setting since it only exploits cache misses.

In the known plaintext attack (Section 2.5), the analysis of the cache hits has to be skipped in order not to exclude the correct hypothesis. This adaptation to a partially pre-loaded cache is fully compatible with our error resilient strategy (Section 2.6.3), though requiring a higher number of inputs. The figures of our simulations are plotted below for different values of these two "real-life" parameters.

2.6.5 Simulation results

In Figure 2.6, we present the results of our simulated attacks against AES-128 for different values of the uncertainty rate (along the x-axis) and the number of preloaded lines (different colors on the graph) with 10^4 random 16-byte keys in each of the 20 cases.

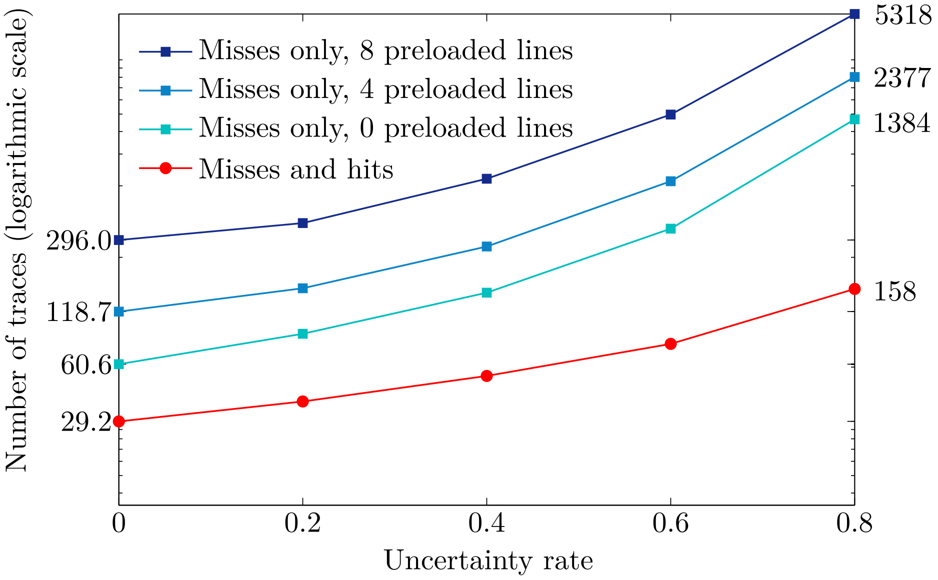


Figure 2.6: Average number of traces required for the full AES key recovery

In these simulations, we gave the same uncertainty rate to the 19 analysed lookups in a cache trace. We believe it accurately models the reality since all cache lookups of a round are performed with the same `load` instruction. If it were not the case, it would be clear that a more uncertainty-prone lookup would represent a bottleneck of the key recovery algorithm.

As we can see, the presence of errors is well tolerated as well the partial preloading of the cache. In the first case, when only 20% of the events are firmly identified, the attack requires less than 160 acquisitions on average. In the second, when half of the lookup table is pre-loaded, an average number of 300 of inputs are required. The combination of these two scenarios highly increases the measurement complexity, yet within a feasible range.

2.7 Countermeasures

In this section we discuss the efficiency against trace-driven cache-collision attacks of countermeasures commonly applied to AES embedded implementations. We also present specific countermeasures that can be adopted to thwart these attacks.

Countermeasures against power analysis attacks have either one or the other objective: masking and hiding. The goal of a masking countermeasure is to make the power consumption independent from the intermediate value, whereas a hiding countermeasure may be the obfuscation of the implementation details in time by shuffling the operations and inserting random delays. A combination of masking and hiding countermeasures provides the best protection against DPA [56].

2.7.1 Masking

First-order masking applies one random map (thus unknown to the attacker) to every sensitive variable (i.e. which depends on the key) so that even if a masked value is recovered, no secret information is disclosed. Therefore, first-order masking schemes provide resistance against first-order DPA attacks.

This *secret sharing* scheme can also be extended to the general case, where the sensitive information is concealed within $n + 1$ shares. The knowledge of up to n shares cannot reveal a single bit of the secret. Therefore, such a scheme is in principle resistant to n-th order DPA attacks.

There exist different types of masking schemes, including Boolean, arithmetic and multiplicative masking. Some schemes suit more some operations, but conveniently, the AES only needs Boolean masking for all of its operations. Several random values, called masks, are required so that the security of the scheme is not exposed when two consecutive values are XORed. Herbst et al. [56] have shown that 6 random masks are sufficient to provide an efficient first-order DPA resistance to a smart card implementation of AES.

The S-box is a non-linear operation. Thus, $S(x \oplus m) \neq S(x) \oplus S(m)$. In order to keep track of the mask value, so as to be able to remove it after encryption, a masked table S' has to be computed for a pair (m, m') of input and output masks, such that: $S'(x \oplus m) = S(x) \oplus m'$. Therefore, when implementing the S-box with lookup tables, one pair of input and output masks implies the computation of one lookup table.

Before each encryption run, random masks are generated and the corresponding masked lookup table S' is precomputed. On constrained devices such as smart cards, a designer will generally keep the complexity of the AES implementation low enough with the pre-computation of a single masked lookup table S'. Therefore, there is a single mask value m XORed with every input value x of the `SubBytes` operation. Against standard DPA, this does not represent a threat because such input values are not added with each other, thus their masks do not cancel out. Nevertheless, it does represent a threat against trace-driven cache-collision attacks because such adversary artificially builds XOR relations between S-box inputs. In these relations, the masks cancel each other out which makes the equations and inequations of the attack remain valid. We verified this reasoning with simulations: the attack success is identical as with a plain AES code.

2.7.2 Hiding

In time dimension, hiding countermeasures randomize the execution flow of an algorithm by shuffling the order in which operations are performed or by inserting random delays or dummy operations.

Shuffling

Following our description of a trace-driven cache-collision attack, it is always assumed that the adversary knows the order in which the S-box lookups are performed. In the first round of AES, 16! possible permutations can be applied to the order of the S-box lookups. If we also consider the two first lookups of the second round, and assuming that a new permutation is applied, we can also multiply the number of possible permutations by 16×15. As a consequence, the application of a random permutation to the order of the S-box lookups renders a trace-driven cache-collision attack practically impossible, at the cost of the generation of a few random permutations.

Insertion of random delays or dummy operations

Random delays aim at breaking the synchronization of the power or EM traces in a DPA attack, thus increasing the measurement complexity [35]. In the context of SPA, the adversary is assumed to be able to identify the specific operation she is looking for. Hence, if the random delays are easy enough to distinguish from cache hits and cache misses, their insertion within the execution flow of the encryption algorithm is not a good protection against SPA, including trace-driven cache-collision attacks.

2.7.3 Specific countermeasures

Given the high number of S-box lookups required in an AES run (160 in our description), it is very likely that all S-box table lines will be accessed. Hence, a sensible implementation of the cipher would pre-fetch the entire S-box lookup table. In this manner, all cache events occurring during the algorithm execution become cache hits and take the same amount of time, thus an adversary cannot draw any information from them.

A less performance-oriented protected implementation would disable the cache mechanism so as to prevent all microarchitectural leakages induced by the cache. All instructions would take the same amount of time to execute, thus defeating the described attacks.

2.8 Theoretical model

In this section we present a theoretical estimation of the number of traces required for the known plaintext attack.

In the work of Acıiçmez and Koç, a theoretical model was already provided to estimate the number of traces required for the analysis of the 15 cache events in the first round [1]. However, this model did not take into consideration the dependency between the cache events and assumed an error-free cache event detection.

First, we describe a sound model to obtain the number of traces required for the error-tolerant analysis of each of the cache events *individually*, both for the first and second rounds. This model takes error detection into consideration. Then, we show that this model, being univariate, still does not estimate the number of traces for the *full* attack precisely due to the statistical dependency between the cache events. The distribution in the multivariate model is however too complex to be expressed theoretically, hence we present some illustrations based on our simulations.

2.8.1 Univariate model for the error-tolerant attack

The model we develop in this section provides an expected number of traces $\mathbf{E}N_i$ required for the analysis of the i-th lookup, $1 \leq i \leq 15$ in the first round and $16 \leq i \leq 19$ in the second round (i.e. the enumeration of the cache events continues in the second round, so as to simplify the formulae), for arbitrary uncertainty rate ρ (per lookup) and number of cache lines per S-box lookup table m. We will implicitly assume that the inputs to the second round lookups are statistically independent of the inputs to the first round lookups. Strictly speaking, this is not true. However, the statistical dependency in this case is not significant and so can be omitted for practical reasons; this is verified by the empirical results that we obtain running attack simulations.

We start with obtaining the expectation $\mathbf{E}R_i$ for the fraction of candidates R_i remaining after analysing lookup i of a single cache trace. This expectation is expressed as

$$\mathbf{E}R_i = \sum_{s=1}^{i} \Pr(T_i = s) \cdot R_i^{(s)}, \quad i \geq 1 \tag{2.12}$$

where T_k is the number of lookup table lines in cache (i.e. $|\Gamma|$) after k lookups, and $R_i^{(s)}$ is the fraction of the key candidates remaining after analysis of the i-th lookup of a single cache trace when the number of lines previously loaded into cache is s. Note that the expression (2.12) works for the second round lookups as well.

The distribution $\Pr(T_k = s)$ is a classical allocation problem [67]. We have:

$$\Pr(T_k = s) = \binom{m}{m-s} \left(\frac{s}{m}\right)^k \Pr_k^0(s) \tag{2.13}$$

where

$$\Pr_k^0(s) = \sum_{l=0}^{s} \binom{s}{l} (-1)^l \left(1 - \frac{l}{s}\right)^k$$

Note that this distribution describes the process within the device and not the attacker's observations and therefore does not depend on the error probability ρ.

The fraction $R_i^{(s)}$ is expressed as the sum of the products of the conditional probabilities of the three possible cache event observations (miss, hit, uncertain) with the corresponding remaining fractions $R_{i,M}^{(s)}$, $R_{i,H}^{(s)}$, $R_{i,U}^{(s)}$) (so, strictly speaking, this fraction is the expected value under a fixed s):

$$\begin{aligned} R_i^{(s)} \; = \; & \Pr(CT_i = M \mid T_i = s) \cdot R_{i,M}^{(s)} \\ & + \Pr(CT_i = H \mid T_i = s) \cdot R_{i,H}^{(s)} \\ & + \Pr(CT_i = U) \cdot R_{i,U}^{(s)} \end{aligned} \tag{2.14}$$

where the three probabilities are expressed as follows:

$$\begin{aligned} \Pr(CT_i = M \mid T_i = s) &= \frac{(1-\rho)(m-s)}{m} \\ \Pr(CT_i = H \mid T_i = s) &= \frac{(1-\rho)\, s}{m} \\ \Pr(CT_i = U) &= \rho \end{aligned}$$

and, recalling the error-tolerant attack description in Section 2.6.3, the fractions for the cases of a miss, a hit and an uncertain event are expressed as:

$$\begin{aligned} R_{i,M}^{(s)} &= \frac{m - (1-\rho)\, s}{m} \\ R_{i,H}^{(s)} &= \frac{s + \rho\, (i-s)}{m} \\ R_{i,U}^{(s)} &= 1 \end{aligned}$$

Finally, from Equations (2.12), (2.13) and (2.14) we can obtain $\mathbf{E}R_i$ for $1 \leq i \leq 19$, i.e. both for the first and second round round lookups.

Knowing $\mathbf{E}R_i$, we can estimate the expected number of traces $\mathbf{E}N_i$ required for the analysis of an i-th lookup. We recall from Section 2.5.1 that in the analysis of each lookup in the first round ($1 \leq i \leq 15$), we want to reduce the number of candidates for the corresponding XOR difference of the key nibbles from m to 1. The traces are statistically independent if the inputs are independent (which is the assumption of our known plaintext attack), and each trace leaves us with a fraction $\mathbf{E}R_i$ of the remaining candidates, so we have:

$$\begin{aligned} m \cdot (\mathbf{E}R_i)^{N_i} \leq 1 \,, \\ \mathbf{E}N_i \approx -\log_{\mathbf{E}R_i} m \,. \end{aligned}$$

We obtain $\mathbf{E}R_i$ and estimate $\mathbf{E}N_i$ for all the lookups of the first round being analysed assuming $m = 16$. The values for the case there are no errors in detection, i.e. $\rho = 0$, are shown in Table 2.2. Note that these values are larger than the ones obtained by Acıiçmez and Koç [1] (denoted there by $R_{expected}^{k}$, k being i in our terms) since in our model we have correctly considered the dependency of the analysis

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathbf{E}R_i$	0.882813	0.787598	0.711151	0.650698	0.603836	0.568490	0.542866	0.525418	0.514812	0.509903	0.509705	0.513373	0.520184	0.529519	0.540853
$\mathbf{E}N_i$	22.24	11.61	8.13	6.45	5.50	4.91	4.54	4.31	4.18	4.12	4.11	4.16	4.24	4.36	4.51

Table 2.2: Expected ratios of the remaining candidates and expected numbers of traces for the first round lookups, $m = 16$, $\rho = 0$.

of an i-th lookup on the events in the previous i lookups (recall that the lookup enumeration starts from 0).

In Figure 2.7, we compare the theoretical estimates for the first round events with the empirical results that we obtained in attack simulations for the cases $\rho = 0$, $\rho = 0.25$ and $\rho = 0.5$. One can see that our model captures well the behaviour of the attack and the effect of the errors.

In the theoretical figures as well as in the empirical ones, the average number of required inputs quickly decrease in the first lookups as we move to the next one, and tend to increase in the last lookups. This is due to the fact that at the beginning of the round, cache hits, more informative than cache misses for small values of s, are less likely to appear and cache misses are more frequent but reduce the set of hypothesis more slowly than cache hits. Around the 12th lookup (the cache is half-filled), cache hits and misses are equally informative and likely to appear, and give the quickest reduction of hypothesis. At the end of the round, cache hits are more frequent but less likely to reduce the set of hypothesis, and reciprocally for cache misses, which increase the number of required inputs.

Similarly, for the lookups of the second round ($16 \leq i \leq 19$) the numbers of traces can be estimated from the inequality

$$z_i \cdot (\mathbf{E}R_i)^{N_i} \leq 1 \,,$$

where z_i is the initial number of candidates in the system of (in)equations for the lookup i. From Section 2.5.2 we recall that $z_{16} = 2^{24}$, $z_{17} = 2^{16}$, $z_{18} = 2^{12}$ and $z_{19} = 2^{16}$. The theoretical estimates of $\mathbf{E}R_i$ and $\mathbf{E}N_i$ for the second round, for the case $m = 16$, $\rho = 0$, are given in Table 2.3. The numbers of traces observed in our experiments are correspondingly 26.86, 19.47, 15.02 and 20.36, thus perfectly matching the theoretical figures.

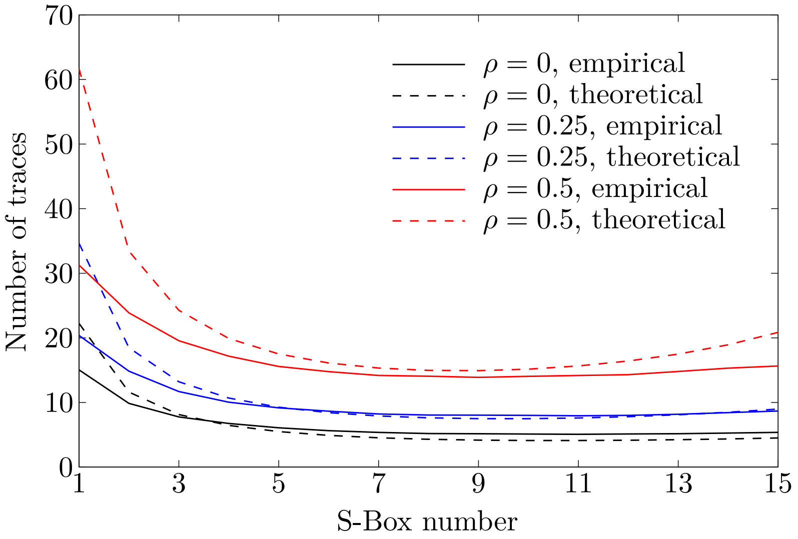


Figure 2.7: Expected number of traces for the lookups of the first round. Theoretical and empirical figures for different error probabilities ρ are shown.

i	16	17	18	19
$\mathbf{E}R_i$	0.553737	0.567792	0.582699	0.598187
$\mathbf{E}N_i$	28.15	19.59	15.40	21.58

Table 2.3: Theoretical estimates for the second round lookups, $m = 16$, $\rho = 0$

The model is also easily adjustable for the miss-only analysis in the case of partially preloaded cache.

2.8.2 Multivariate model

The full attack measurement complexity is determined by the maximum number of traces required for the analysis of each lookup. If the cache events were statistically independent, the expectation for the maximum number of traces $\mathbf{E}(\max_i N_i)$ would be equal to the maximal expected number of traces among each of the lookups $\max_i(\mathbf{E}N_i)$, so one could use the model described above.

However, the cache events are dependent, therefore:

$$\mathbf{E}(\max_i N_i) \neq \max_i(\mathbf{E}N_i) \, ,$$

and so the univariate model is not applicable for the estimation of the full attack complexity. This is well shown by the results of attack simulations: for example, in case $\rho = 0$ for the first round in the univariate model we observed $\max_i(\mathbf{E}N_i) = \mathbf{E}N_1 = 15$, whereas $\mathbf{E}(\max_i N_i) = 19$. The same holds for the second round, though not as explicit: the observed average maximum number of traces for the *full* analysis of the second round is 29, whereas the observed maximal number of traces for an *individual* lookup is 27.

Therefore, to estimate the number of traces required for the full attack, one has to consider the distribution of $\max_i N_i$ for the case of statistically dependent random variables N_i, which requires the multivariate distribution for $(N_1, N_2, \ldots, N_{19})$. This multivariate distribution is hard to express analytically, it is easier to simulate it carrying out the attack and sampling the values of $\max_i N_i$. This is what we do to obtain the results presented in Section 2.6.5.

To better illustrate the behaviour of the attack, in Figure 2.8a we present the empirical bivariate distribution for the case of the second and third lookups in the first round. In Figure 2.8b we show the distribution for $\max(N_1, N_2)$ against the independent distributions for N_1 and N_2. The mean of the former is 18.04. One can see that it is greater than the maximum of the means for N_1 and N_2 and is actually quite close to the number of traces required for the analysis of all the 15 lookups in the first round.

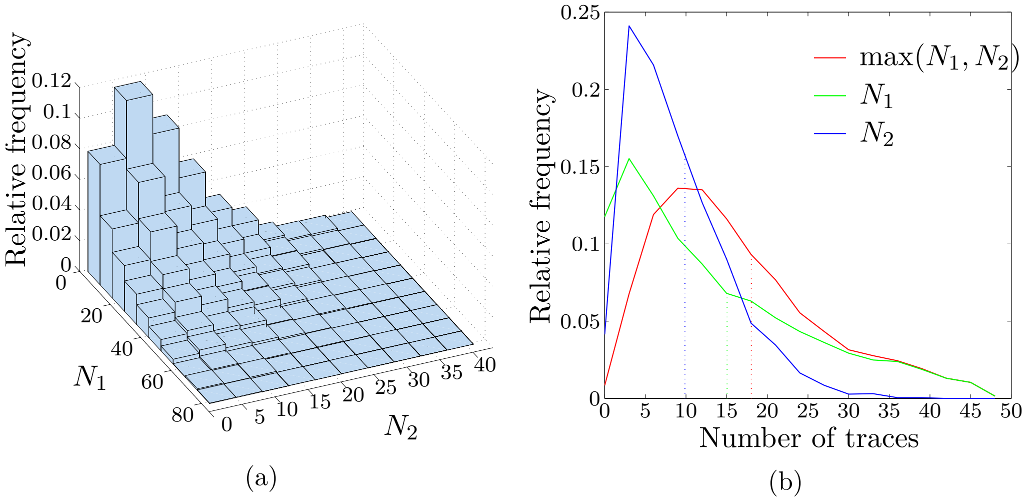


Figure 2.8: (a) empirical bivariate distribution for (N_1, N_2); (b) empirical univariate distributions for N_1, N_2, and for $\max(N_1, N_2)$, dashed lines showing the corresponding means

Nevertheless, the univariate model provides an estimate for the lower bound for the number of traces required for the full attack, so it can be still applied when one requires this bound.

2.9 Conclusion

In this chapter, we have elaborated on attacks which exploit the cache activity of a device derived from side-channel traces. In particular, we have conducted practical explorations of this microarchitectural leakage with EM measurements performed on a 32-bit ARM microcontroller and have shown clear distinctions between cache hits and cache misses.

Targeting AES embedded implementations for such devices, we have improved the chosen plaintext strategy of Fournier and Tunstall against [45] and we have devised a known plaintext approach. We have shown that monitoring the cache activity of a device of about 30 AES encryptions of known plaintexts by the mean of electromagnetic measurements can allow an adversary to recover the entire key when the S-box is implemented as a 256-byte lookup table. We have adapted our attacks so that they can tolerate the presence of noise in the measurements and S-box data in the cache prior to the encryption. We have presented the theoretical model of the attacks.

We discussed the relevance of common DPA countermeasures applied to embedded implementations of AES and concluded that certain Boolean masking techniques and the insertion of random delays have no effect to prevent trace-driven cache-collision attacks. As a result, trace-driven attacks can defeat protected implementations requiring the encryption of about 30 known plaintexts, where second-order DPA would typically take thousands of acquisitions. However, simple and lightweight countermeasures such as lookup shuffling or pre-caching of the entire S-box table make the attack practically infeasible. The offline complexity, i.e. the execution of the sieve, takes a few seconds on a standard PC.

Chapter 3

Side-channel analysis of the modular addition

The modular addition is present in numerous symmetric schemes, in particular in ARX constructions. This operation is in essence more resistant to DPA attacks than S-boxes and as a result, standard DPA fails to recover all the key bits. We put forward a generic and practical approach that is applicable against many constructions: we choose a more complex target function such as the combination of two modular additions. To deal with the high number of key bits involved, we apply a further divide and conquer approach. The number of key hypothesis consequently remains within a feasible range at the expense of the amplitude of the correlation coefficients. We verify the validity of our approach with experiments on an 8-bit AVR microcontroller. Additionally, we carry out the recovery of the entire key from a recommended implementation of Threefish on the same board.

This is a joint work with Arnab Roy and Praveen Kumar Vadnala, published in the proceedings of WESS 2012 [115].

Contents

3.1 Introduction

In a DPA attack, three critical parameters have to be considered: the target operation, the power model and the statistical distinguisher. They need to be chosen according to the cipher under attack along with its implementation details and the target device architecture. Note that in practice, the attention brought to the measurement setup is also crucial and in particular, proper values for the device clock frequency and the DSO sampling rate can help reduce the various amounts of noise present in the power or EM traces.

When a block cipher algorithm features a key-dependent S-box, its output is a natural target for a DPA attack. Indeed, the role of an S-box is precisely to provide confusion through its non-linearity so as to improve the resistance of the block cipher against linear and differential cryptanalysis. However, Prouff gave a mathematical evidence that the more non-linear an S-box is, the more it helps a side-channel adversary to recover the key [88]. Besides the prevalence of DES and AES in symmetric cryptographic applications nowadays, another reason why a majority of published DPA attacks target these ciphers is that S-box-based block ciphers clearly illustrate the efficiency of DPA attacks. In an attempt to thwart these attacks, the S-box operation can be implemented in a resistant way against first- or higher-order DPA attacks in software through masking [82] or in hardware through secure logic style [111].

Benoît and Peyrin simulated DPA attacks in the Hamming weight power model and using the correlation coefficient as the statistical distinguisher against six SHA-3 candidates [11]. In accordance with Prouff, they showed that the correct key used in an AES S-box—highly non-linear—, is more easily distinguishable from the wrong key hypothesis than in a modular addition—less non-linear than the AES S-box. The XOR operation—which is linear—, leads to higher correlation coefficients for wrong keys hypothesis and is therefore the most difficult operation to retrieve the correct key from in a DPA attack scenario.

But as previously mentioned, a high confusion is an absolute necessity for a block cipher and attention has recently been given to alternatives to S-box-based block ciphers. ARX constructions for example are composed of three operations: modular addition, bit rotation and exclusive-or, the only non-linear one being the modular addition. They are fast in both software and hardware, hence the weak non-linearity of the modular addition can be compensated with a high number of rounds.

In a side-channel attack against an ARX block cipher or hash function (as used in a MAC algorithm), the natural target operation is the modular addition because it is the only non-linear operation. Practical attacks in the Hamming weight power model have been reported. Lemke et al. against IDEA [70] and Boura et al. against Skein [25] provide ambivalent conclusions: standard DPA does not always output the correct key, regardless of the measurement complexity. Against Skein, Zohner et al. exploit the particular distribution of the correlation coefficients among the key candidates in the Hamming weight power model to gain information about the correct key [125, 126].

The scope of this work is the study of the side-channel resistance of the modular addition and the application of DPA attacks against this operation.

3.1.1 Contributions

In this work, we apply practical and generic methods to circumvent the difficulties that arise in a DPA attack on modular addition. Our experiments are carried out on an AVR 8-bit microcontroller with a typical DPA measurement setup. Our methods are tested against a fast implementation of the Threefish block cipher for AVR microcontrollers recommended by the designers of Skein.

3.1.2 Terminology

Throughout this chapter, the operations are performed on blocks of b bits and modular additions are modulo 2^b. The key is represented as a concatenation of blocks: $k = k_1 \| k_2 \| \dots \| k_{\#_b}$, where $\|$ is the concatenation operator and $\#_b$ is the number of blocks. Similarly, the plaintext is denoted by $p = p_1 \| p_2 \| \dots \| p_{\#_b}$. Thus, the key (resp. plaintext) size is $\#_b \cdot b$. The word length of the device is w bits, and we sensibly assume $w \leq b$. The number of registers required to store each block is thus $\#_w = \frac{b}{w}$. When required, we will also include the register count as second index so that a key (resp. plaintext) block is represented as: $k_i = k_{i,1} \| k_{i,2} \| \dots \| k_{i,\#_w}$ (resp. $p_i = p_{i,1} \| p_{i,2} \| \dots \| p_{i,\#_w}$). When truncating further the key and plaintext words (as we do in Section 3.4.2), the key (resp. plaintext) word is divided into $\#_d$ chunks of length $d = \frac{w}{\#_d}$ and the chunk count written as third index, so that a key (resp. plaintext) word is represented as $k_{i_b,i_w} = k_{i_b,i_w,1} \| k_{i_b,i_w,2} \| \dots \| k_{i_b,i_w,\#_d}$.

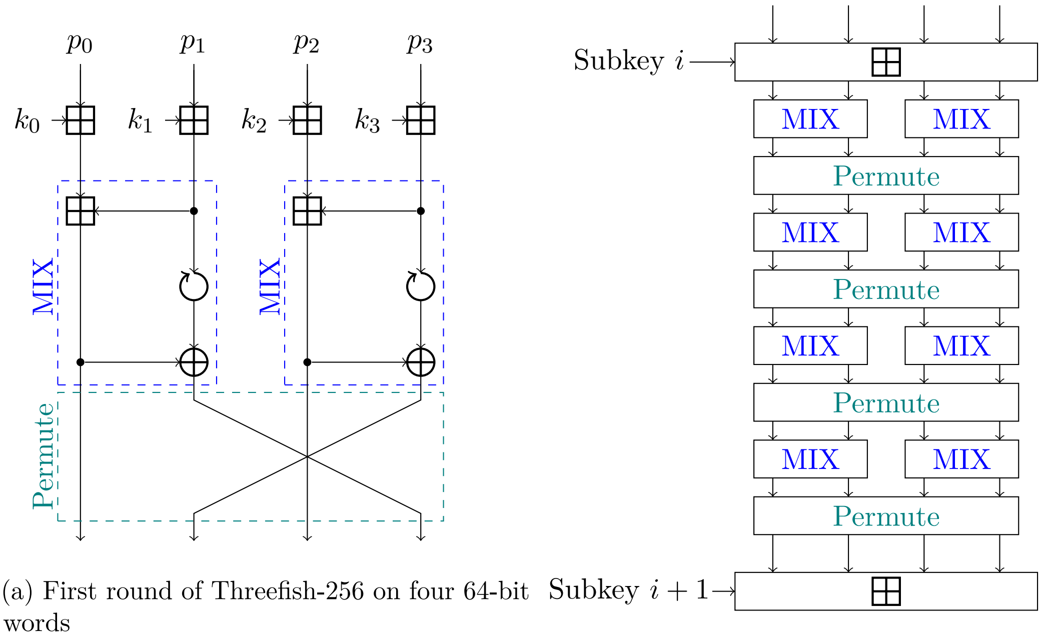
3.2 Background

In this section we describe Threefish, the block cipher on which we carry out our experiments (Section 3.6).

3.2.1 The Threefish block cipher

The block cipher Threefish is at the core of the Skein hash function [40], which was one of the five finalists of the SHA-3 competition launched by the American National Institute of Standards and Technology in the quest for a new hash function standard. Threefish is tweakable, which means that it takes an extra parameter called the *tweak* along with the key and the plaintext or ciphertext. This aims at providing additional security to a hash function when the latter is based on a block cipher. Threefish is only composed of the ARX operations: modular addition, rotation and exclusive-or. These simple operations are efficiently implemented in both software and hardware, which allows for a higher number of rounds in the block cipher construction in comparison with block ciphers using S-boxes (AES has 10, 12 or 14 rounds depending on the key length)—Threefish takes 72 to 80 rounds. In the meantime, the high number of rounds in an ARX block cipher is necessary to achieve a good diffusion and non-linearity.

Threefish has three available block sizes: 256, 512 and 1024 bits. Threefish-256, the algorithm we target in the application of our attacks in Section 3.6, is ideally suited for constrained devices such as our 8-bit AVR ATMega128 microcontroller. The key length is the same as the block size and the tweak is 128 bits long, irrespective of the size of the block. The Threefish operations are performed on 64-bit words.



(a) First round of Threefish-256 on four 64-bit words

(b) Four rounds of Threefish-256

Figure 3.1: Structure of Threefish-256

The subkeys are obtained from the master key and the tweak via the key schedule (which we do not describe here). Every four rounds, new subkeys are added to the 64-bit states (initialized to the plaintext words). A round transformation consists of a MIX function, composed of a modular addition, a bit-rotation and an XOR; a permutation of the words is then applied. In Threefish-256, 72 round transformations are executed, involving the master key plus 18 subkeys.

The non-linearity of Threefish comes from the modular addition in the MIX function, more precisely from the produced carry bits. This non-linearity helps a side-channel adversary to identify the correct key from the correlation, but in the meantime the carry bits leads to discrepancies in the key recovery, as described in the next section.

3.3 Attacks on the modular addition

In this section, we describe some general considerations of a DPA attack against modular addition.

DPA attacks targeting arithmetic operations were theoretically considered during the AES competition [30] and also within the New European Schemes for Signatures, Integrity and Encryption (NESSIE) project [83]. Practical attacks on the XOR, the modular multiplication and the modular addition were first conducted by Lemke et al. [70]. Concerning the modular operation, they observed a singularity in the ranking of the key candidates in their simulations in the Hamming weight model: although the correct key hypothesis leads as expected to the highest correlation in absolute value, the key hypothesis differing from the correct key only by its most significant bit (MSBi) is ranked second, the two key hypothesis differing from the first and second candidates only by the two MSBis are ranked third and fourth, and so on.

We confirm the observation of this phenomenon with the computation of the correlation coefficients for the byte modular addition between the Hamming weights of intermediate values depending on the correct key on the one hand, and on all key hypothesis on the other hand. In Figure 3.2, the top four key candidates are indeed, in order of decreasing correlation:

1. the correct key $k = 50 =$ `0b00110010`,
2. $k + 128 =$ `0b10110010`
3.
4. $\left.\right\}$ $k + 64 = 114 =$ `0b01110010` and $k - 64 \equiv 242 \pmod{256} =$ `0b11110010`

The correlation corresponding to the other candidates follows the same classification with respect to their MSBis. This particular distribution of the correlation coefficients among the key hypothesis can be explained by the carry bit propagation. If a carry difference between an hypothesis and the correct intermediate value occurs at the MSBi, it will have no effect on the bits under attack, but on the the subsequent ones. On the opposite, if a carry difference occurs at the least significant bit (LSBi) of the intermediate value, the carry effect will be propagated on a number of the bits under attack.

A consequence of this classification of the correlation coefficients among the key candidates is the symmetry of the distribution. As we observe in Figure 3.2, the correlation coefficients are symmetric with respect to the correct key k and its counterpart differing from its MSBi, respectively 50 and 178 in the figure. Algebraically, these symmetries are expressed as:

$$r(k + 128i - j \pmod{256}) = r(k + 128i + j \pmod{256}), \quad i, j \in \mathbb{Z},$$

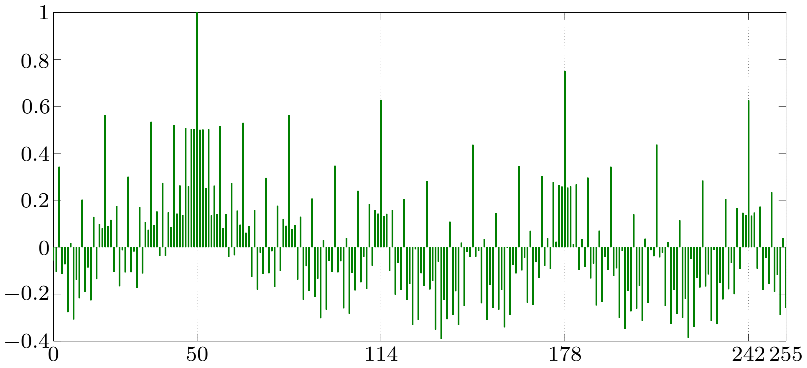


Figure 3.2: Simulated correlation coefficients for all key hypothesis involved in a modular addition. The correct key value is 50.

where $r(k^*)$ is Pearson's correlation coefficient computed for the Hamming weight of the intermediate value for an hypothetical key value k^* and the actual key value k: $r(s) = \rho(W_H(k^* + M \pmod{2^8}), W_H(k + M \pmod{2^8}))$ and M is the plaintext random variable taken from a uniform distribution.

3.3.1 Practical attacks on single modular addition

Boura et al. applied conventional correlation-based DPA techniques in the Hamming weight model against Skein in MAC constructions and encountered the same discrepancies in the key recovery [25].

In Section 3.5 we provide the results of our own experiments and confirm the failure of the recovery of some of the key bytes.

3.3.2 The butterfly attack

Based on the observation that the correct key k is the point of origin of symmetric correlation coefficients, as described by the relation:

$$r(k - j \pmod{256}) = r(k + j \pmod{256}), \quad j \in \mathbb{Z},$$

Zohner et al. proposed an attack which exploits this inherent property of the modular addition to identify the correct key with higher success [125]. Their so-called butterfly attack computes the intermediate values based on key hypothesis, applies the Hamming weight as power model and computes the correlation coefficients, thus following a conventional DPA approach. Then, the butterfly attack executes an additional step which aims at determining the point of origin of the most accurate symmetry in the correlation coefficients with a least square approach. Namely, for each key candidate k^*, the method accumulates the square of the Euclidean distance between the correlations of k^* and any other candidate j:

$$\text{lsq}(k^*) = \sum_{j=1}^{255} (r(k^* + j \pmod{256}) - r(k^* - j \pmod{256}))^2.$$

Clearly, in a noise-free environment, the least square approach returns 0 for $k^* = k$ because the symmetry of the correlation coefficients is centred in k. However, $k^* = k + 128 \pmod{256}$ is also a point of origin of the symmetry of the correlation coefficients, therefore its sum of squared distances is also 0. The butterfly attack applied to our simulated data clearly outputs two top candidates, as depicted in Figure 3.3.

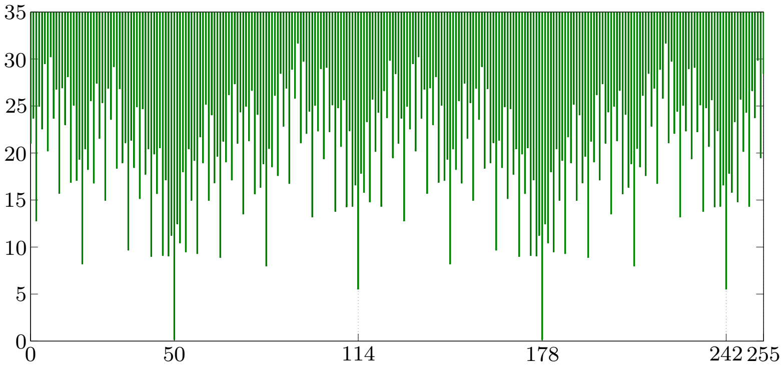


Figure 3.3: Butterfly attack applied to simulated (noise-free) data. The correct key value is 50.

The least square approach is an additional step to correlation-based DPA attacks against modular addition which efficiently reduces to 2 the number of key candidates—200 power measurements on an 8-bit AVR ATMega 2561 microcontroller were necessary. However, this duplicity of candidates on key chunks can be problematic against constructions with large block sizes: Threefish-256 would eventually take an exhaustive search over 2^{32} full key candidates and Skein-1024 up to 2^{128} which is clearly infeasible. Note that sorting algorithms like the ones presented in Chapter 5 will not be relevant since the probabilities of the top two candidates are strictly the same. Zohner et al. put forward the overlapping of key bits in two target key chunks to circumvent this difficulty at the cost of a higher number of key chunks—thus yielding a DPA attack of higher offline complexity.

The approach we present in Section 3.4 is more generic because it applies to a broader class of devices, not only devices that leak in the Hamming weight model.

3.4 Practical approaches

In this section we propose practical approaches for a DPA attack against modular addition. Similar strategies have been applied for key recovery against AES hardware implementations by Batina et al. [8] and Kizhvatov [64].

We first describe a standard DPA attack against modular addition. Let us consider the following operation:

$$r \leftarrow p \boxplus k \tag{3.1}$$

where the result of the addition of the plaintext p and the key k modulo 2^n are stored in the register r. The plaintexts p are chosen from a random enough distribution and the power consumption induced by this operation on the target device is acquired. Then the results of the modular addition are predicted over all key candidates and these intermediate values are mapped to power consumption values using a power model. Finally, for every key candidate the predicted power consumption is confronted with the actual one through a statistical distinguisher. The candidate which matches best the power consumption of the target device is expected to be the correct key.

However, this straightforward approach fails to recover all the bytes used in a modular addition, as shown by previous attacks in the Hamming weight power model (Section 3.3) and by our own experiments (Section 3.5), using the Hamming distance as power model. Namely, we were able to recover correctly only 48 out of the 64 bits of the key.

As previously mentioned, the leakage induced by a single modular addition is not enough to recover the entire key. However, subsequent operations in the cipher may also leak information about the key. The attacker can choose to exploit a sequence of instructions instead of a single one so as to capture more relevant leakage. For example, let us consider as in Figure 3.4 the following two modular additions: $p_i \boxplus k_i$ and $p_j \boxplus k_j$, and the combination of the corresponding instructions within a third one: $(p_i \boxplus k_i) \odot (p_j \boxplus k_j)$, where the $\odot$ operator can either refer to the modular addition or the XOR.

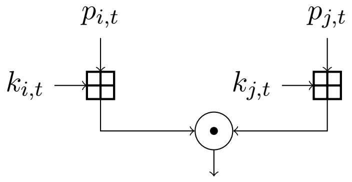


Figure 3.4: Application of operation $\odot$ on the result of two modular additions involving two key blocks.

At the instruction level, assuming that the registers r_a and r_b are initialized to p_i and p_j respectively, three steps are performed:

$$r_a \leftarrow r_a \boxplus k_i$$
$$r_b \leftarrow r_b \boxplus k_j$$
$$r_a \leftarrow r_a \odot r_b$$

From attacking one to three combined operations, the number of key bits to recover is doubled, which quadratically increases the number of hypothesis to compute and thus, the cost of the offline phase of the attack. However, such attack still falls into the class of first-order DPA attacks since a single instance of time is considered, unlike second-order DPA for example where preprocessing of the power traces is necessary.

The success of a DPA attack against this combined modular additions depend on the nature of the combining $\odot$ operation.

Such combination of two modular additions is found in various block ciphers, such as TEA [119], FEAL [68] and Threefish [40], on which we test our methods in Section 3.6.

Algorithm 6 Computation of an n-bit output by successive application of modular addition followed by $\odot$ on $2n$-bit plaintext and key

```
Input: Plaintext blocks: $p_i$ and $p_j$ — Key blocks: $k_i$ and $k_j$.
1: $Out \leftarrow zeros(n)$
2: $Interm \leftarrow zeros(2n)$
3: $Interm(1:n) \leftarrow p_i \boxplus k_i$
4: $Interm(n+1:2n) \leftarrow p_j \boxplus k_j$
5: $Out(1:n) \leftarrow Interm(1:n) \odot Interm(n+1:2n)$
Output: $(p_i \boxplus k_i) \odot (p_j \boxplus k_j)$
```

3.4.1 First approach: Guessing two blocks at a time

We now describe the successive states of the registers during the execution of Algorithm 6, where two key blocks p_i and p_j are added with two key blocks k_i and k_j.

The registers are initially configured as follows:

$$\begin{aligned}
r_1 &\leftarrow p_{i,1} & r_{\#_w+1} &\leftarrow p_{j,1}\\
r_2 &\leftarrow p_{i,2} & r_{\#_w+2} &\leftarrow p_{j,2}\\
&\vdots & &\vdots\\
r_{\#_w-1} &\leftarrow p_{i,\#_w-1} & r_{2\#_{w-1}} &\leftarrow p_{j,\#_w-1}\\
r_{\#_w} &\leftarrow p_{i,\#_w} & r_{2\#_w} &\leftarrow p_{j,\#_w}
\end{aligned}$$

During the execution of the modular addition instruction, i.e. steps 3 and 4 of Algorithm 6, the contents of the registers are updated in this way:

$$\begin{aligned}
r_1 &\leftarrow p_{i,1} \boxplus k_{i,1} & r_{\#_w+1} &\leftarrow p_{j,1} \boxplus k_{j,1}\\
r_2 &\leftarrow p_{i,2} \boxplus k_{i,2} \boxplus c_{i,1} & r_{\#_w+2} &\leftarrow p_{j,2} \boxplus k_{j,2} \boxplus c_{j,1}\\
&\vdots & &\vdots\\
r_{\#_w-1} &\leftarrow p_{i,\#_w-1} \boxplus k_{i,\#_w-1} \boxplus c_{i,\#_w-2} & r_{2\#_w-1} &\leftarrow p_{j,\#_w-1} \boxplus k_{j,\#_w-1} \boxplus c_{j,\#_w-2}\\
r_{\#_w} &\leftarrow p_{i,\#_w} \boxplus k_{i,\#_w} \boxplus c_{i,\#_w-1} & r_{2\#_w} &\leftarrow p_{j,\#_w} \boxplus k_{j,\#_w} \boxplus c_{j,\#_w-1}
\end{aligned} \tag{3.2}$$

where $c_{i,t}$ refers to the carry produced from the word t of block i and the modular addition $\boxplus$ is modulo 256.

Note that here the carries produced by the additions performed inside the block are considered, but the addition at the block level is modulo 2^n. Finally during the application of step 5 of the algorithm, the registers are configured in the following way:

$$\begin{aligned}
r_1 &\leftarrow (p_{i,1} \boxplus k_{i,1}) \odot (p_{j,1} \boxplus k_{j,1})\\
r_2 &\leftarrow (p_{i,2} \boxplus k_{i,2} \boxplus c_{i,1}) \odot (p_{j,2} \boxplus k_{j,2} \boxplus c_{j,1})\\
&\vdots\\
r_{N_r-1} &\leftarrow (p_{i,N_r-1} \boxplus k_{i,N_r-1} \boxplus c_{i,N_r-2}) \odot (p_{j,N_r-1} \boxplus k_{j,N_r-1} \boxplus c_{j,N_r-2})\\
r_{\#_w} &\leftarrow (p_{i,\#_w} \boxplus k_{i,\#_w} \boxplus c_{i,\#_w-1}) \odot (p_{j,\#_w} \boxplus k_{j,\#_w} \boxplus c_{j,\#_w-1})
\end{aligned}$$

Assuming that the device leaks in the Hamming distance model, we map these intermediate values in the registers after step 5 of Algorithm 6 to the following leakages values as follows:

$$\begin{aligned}
l_1 &= W_H((p_{i,1} \boxplus k_{i,1}) \oplus ((p_{i,1} \boxplus k_{i,1}) \odot (p_{j,1} \boxplus k_{j,1}))) \qquad (3.3)\\
l_2 &= W_H((p_{i,2} \boxplus k_{i,2} \boxplus c_{i,1}) \oplus ((p_{i,2} \boxplus k_{i,2} \boxplus c_{i,1}) \odot (p_{j,2} \boxplus k_{j,2} \boxplus c_{j,1})))\\
&\vdots\\
l_{\#_w-1} &= W_H((p_{i,\#_w-1} \boxplus k_{i,\#_w-1} \boxplus c_{i,\#_w-2})\\
&\quad \oplus ((p_{i,\#_w-1} \boxplus k_{i,\#_w-1} \boxplus c_{i,\#_w-2}) \odot (p_{j,\#_w-1} \boxplus k_{j,\#_w-1} \boxplus c_{j,\#_w-2})))\\
l_{\#_w} &= W_H((p_{i,\#_w} \boxplus k_{i,\#_w} \boxplus c_{i,\#_w-1})\\
&\quad \oplus ((p_{i,\#_w} \boxplus k_{i,\#_w}) \odot (p_{j,\#_w} \boxplus k_{j,\#_w} \boxplus c_{j,\#_w-1})))
\end{aligned}$$

where $W_H(x)$ refers to the Hamming weight of x.

The key recovery follows a conventional DPA attack approach using the combination of two modular additions as intermediate value. Because of the carry bit involved in the subsequent intermediate values, the attacker proceeds from the least to the most significant words of the block, recovering one word at a time. For each word of intermediate value, the key hypothesis to be computed and checked against the measurements are all possible pairs of key words $k_{i,t}$ and $k_{j,t}$, $1 \leq t \leq \#_w$. The key pair hypothesis which yields the highest correlation coefficient is the correct one, as shown by our experiments in Section 3.5.2.

The correct word for an intermediate value is iteratively recovered, allowing the attacker to predict the two carry bits $c_{i,t}$ and $c_{j,t}$ used in the next word. At the end, all the key words are recovered.

Attack complexity

We describe here the attack complexity in terms of the number of hypothesis considered. For each pair of key words (of length w bits), 2^{2w} key pair hypothesis are considered. As two key blocks are composed in total of $2\#_w$ words, the complexity of the full key recovery is equal to $\#_b\#_w 2^{2w-1}$. As we can see, the attack complexity turns out to be very high for devices with a word length w larger than 16 bits. In the next section, we propose an improvement to our strategy so as to render it practical for such devices.

3.4.2 Approach 2: Using divide and conquer

The modular addition is generally performed bitwise and serially (although more complex adders may feature special improvements, such as the carry look-ahead adder which calculates the carry bits in advance [84]). Since the operation has to be divided depending on the device architecture, the attacker can decide to divide further the number of key bits to be recovered at a time. According to Tunstall et al. [113] and Brier et al. [27], there exists a linear relation between the correlation coefficients ρ_n computed on data of size n bits and the correlation coefficients ρ_m computed on an m-bit truncation of these data, $m \leq n$:

$$\rho_m = \rho_n \sqrt{\frac{m}{n}} \tag{3.4}$$

We use this relation to enhance the feasibility of our attack against devices with large block sizes. We divide each key word k_{i_b,i_w} ($1 \leq i_b \leq \#_b$ and $1 \leq i_w \leq \#_w$) in $\#_d$ chunks of length $d = \frac{w}{\#_d}$ and denote a chunk as: k_{i_b,i_w,i_d} with $1 \leq i_d \leq \#_d$. Hence, the leakages corresponding to the first chunks of two key blocks k_i and k_j in the $\#_w$ registers will be:

$$\begin{align}
l_1 &= W_H((p_{i,1,1} \boxplus k_{i,1,1}) \tag{3.5}\\
&\quad \oplus ((p_{i,1,1} \boxplus k_{i,1,1}) \odot (p_{j,1,1} \boxplus k_{j,1,1}))) \tag{3.6}\\
l_2 &= W_H((p_{i,2,1} \boxplus k_{i,2,1} \boxplus c_{i,1,1}) \notag\\
&\quad \oplus ((p_{i,2,1} \boxplus k_{i,2,1} \boxplus c_{i,1,1}) \notag\\
&\quad \odot (p_{j,2,1} \boxplus k_{j,2,1} \boxplus c_{j,1,1}))) \notag\\
&\vdots \notag\\
l_{\#_w-1} &= W_H((p_{i,\#_w-1,1} \boxplus k_{i,\#_w-1,1} \boxplus c_{i,\#_w-2,1}) \notag\\
&\quad \oplus ((p_{i,\#_w-1,1} \boxplus k_{i,\#_w-1,1} \boxplus c_{i,\#_w-2,1}) \notag\\
&\quad \odot (p_{j,\#_w-1,1} \boxplus k_{j,\#_w-1,1} \boxplus c_{j,\#_w-2,1}))) \notag\\
l_{\#_w} &= W_H((p_{i,\#_w,1} \boxplus k_{i,\#_w,1} \boxplus c_{i,\#_w-1,1}) \notag\\
&\quad \oplus ((p_{i,\#_w,1} \boxplus k_{i,\#_w,1}) \notag\\
&\quad \odot (p_{j,\#_w,1} \boxplus k_{j,\#_w,1} \boxplus c_{j,\#_w-1,1}))) \notag
\end{align}$$

We start by recovering $k_{i,1,1}$ and $k_{j,1,1}$ from the leakage l_1 in Equation (3.5) and then continue with the recovery of $k_{i,2,1}$ and $k_{j,2,1}$ from the leakage l_2 in Equation (3.6) and so on, for each of the $\#_w$ registers. By iteratively applying these steps for each of the $\#_d$ chunks of the words, an attacker is able to recover the entire blocks k_i and k_j. Finally, the iteration of these steps for all the key blocks allow an attacker to recover the entire key.

Attack complexity

As before, we express the complexity of the attack in terms of the number of key chunk hypothesis. A correct pair of key chunks is found among 2^{2d} chunk hypothesis.

Thus, a correct pair of key words is retrieved among $\#_d 2^{2d}$ chunk hypothesis. A correct pair of key blocks k_i and k_j is found among $\#_w \#_d 2^{2d}$ chunk hypothesis and in total, the entire key recovery necessitates $\frac{\#_b}{2} \#_w \#_d 2^{2d}$ hypothesis (the $\frac{1}{2}$ comes from that the key blocks are recovered pairwise).

3.4.3 One block is known

Some ciphers feature a combination of modular addition with a constant, as depicted in Figure 3.4.3. In the block cipher TEA [119] and its variant XTEA [120] for example, it is possible to target the XOR of two modular additions, only one of them involving a key block. The same methods as in the previous sections can be applied, at a lower cost in terms of key hypothesis. Let us denote by ξ the constant value (of same length as the plaintext and key). As before, ξ is divided into $\#_b$ blocks, and each block is further divided into $\#_w$ words, respectively denoted as first and second indexes. This approach is a straightforward correlation-based DPA attack, yet the slightly higher complexity of the intermediate value yields successful key recoveries as we show in Section 3.5.

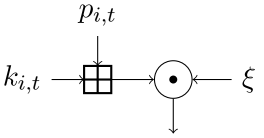


Figure 3.5: Combination of operation $\odot$ on the result of a modular addition and a constant value

Assuming that the device leaks in the Hamming distance power model, the leakages of the registers can be expressed as:

$$\begin{aligned}
l_1 &= W_H((p_{i,1} \boxplus k_{i,1}) \oplus ((p_{i,1} \boxplus k_{i,1}) \odot \xi_{i,1})) \\
l_2 &= W_H((p_{i,2} \boxplus k_{i,2} \boxplus c_{i,1}) \\
&\quad \oplus ((p_{i,2} \boxplus k_{i,2} \boxplus c_{i,1}) \odot \xi_{j,2})) \\
&\vdots \\
l_{\#_w-1} &= W_H((p_{i,\#_w-1} \boxplus k_{i,\#_w-1} \boxplus c_{i,\#_w-2}) \\
&\quad \oplus ((p_{i,\#_w-1} \boxplus k_{i,\#_w-1} \boxplus c_{i,\#_w-2}) \odot \xi_{j,\#_w-1})) \\
l_{\#_w} &= W_H((p_{i,\#_w} \boxplus k_{i,\#_w} \boxplus c_{i,\#_w-1}) \\
&\quad \oplus ((p_{i,\#_w} \boxplus k_{i,\#_w}) \odot \xi_{j,\#_w}))
\end{aligned}$$

The leakage values l_1, computed for every key word hypothesis $k_{i,1}$, is checked against the actual power values, through the measure of the correlation, and so on, for each of the $\#_w$ key words. The same is done for each of the $\#_b$ key blocks, so as to recover the entire key. The complexity of this method can be further reduced by applying the divide and conquer approach described in the previous section.

Attack complexity

The recovery of a key word takes 2^w guesses, the recovery of a key block $\#_w 2^w$ guesses and the recovery of the entire key $\#_b \#_w 2^w$ guesses in total. Similarly, the application of the divide and conquer method will take $\#_b \#_w \#_d 2^d$.

3.5 Experimental results

In this section, we present the results of our experiments in the application of our methods described in Section 3.4. Our experiments were conducted on two different 8-bit RISC-based AVR microcontrollers built based on the Advanced Harvard Architecture. The measurements were taken with a Lecroy Waverunner 104MXi DSO. The power consumption was captured at 500 MS/s.

Our experiments include the recovery of two 64-bit key blocks in a single modular addition; we present both the successful and unsuccessful outcomes (Section 3.5.1). Then we move on to the experiments of a DPA attack against a combination of modular operations for two cases: (1) the combination operation is the modular addition and the power model is the Hamming distance; (2) the combination operation is the XOR and the power model is the Hamming weight (Section 3.5.2). We also present experiments of the divide and conquer approach by recovering parts of the involved key bits (Section 3.5.3). Finally, we show the results of key recovery against a combination of operation where one block is known (Section 3.5.4).

3.5.1 Single modular addition

The key recovery against a single modular addition is set up as follows: the key is 128-bit long, and 5000 plaintexts of same length are randomly generated. The key and plaintexts are divided into two blocks of 64-bit length. The key and plaintext blocks are added modulo 2^{64} while the power consumption of the device leaking the Hamming distance is acquired. The plaintext bytes are stored in 16 registers and the results of the modular addition are stored back in the same registers. Hence, the hypothetical values are chosen as the Hamming distance between a plaintext byte and the corresponding result of the operation. The correlation coefficients are calculated from these hypothetical power values with the power traces. For each of the key blocks, we are able to recover all the key bytes but the first and last (8^{th}) ones, that is, we recover 96 out of the 128 bits of the full key.

The correlation corresponding to the 1^{st} byte of a key block is plotted in Figure 3.6. One can see that the correct key does not yield the highest peak, nor is it ranked in the top ten candidates. The same conclusion applies to the observation of the correlation trace obtained in the recovery of the last byte of a block, as it is plotted in Figure 3.7. For all figures, the black plot corresponds to the correct key.

However, we are successful in recovering the other bytes in a key block. A thousand power traces are enough, since the amplitude of the correlation coefficients does not evolve much with more traces. The amplitude of the highest peak of all candidates in the recovery of the 2^{nd} byte in a key block is plotted in Figure 3.8 in

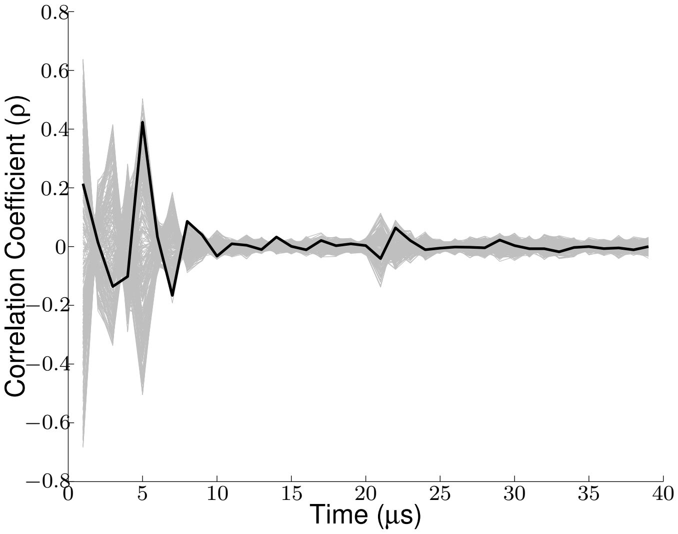


Figure 3.6: Unsuccessful recovery of the first byte in a key block

relation to the number of traces.

3.5.2 Combination of operations

We present below the results of our experiments in the application of the method we describe in Section 3.4.1, that is, targeting the combination of two modular additions.

In a first case, the combination (⊙ in Figure 3.4) is the modular addition and the device under attack has shown to leak in the Hamming distance power model. Therefore, the correlation traces are calculated based on the Hamming distance between the result of the first addition and the third one. In this scenario, all the 128 bits of the key are successfully recovered. In Figure 3.9a, the correlation trace corresponding to the correct key is quite easily distinguishable from the other traces. As shown in Figure 3.9b, 1500 traces are enough to identify the correct key.

In a second case, we apply this approach when the combination of two modular additions is the XOR and the device under attack leaks in the Hamming weight power model. As shown by the correlation traces in Figure 3.10, the pair of first bytes in two key blocks is successfully recovered. The subsequent bytes in a block yield comparable success.

3.5.3 Divide and conquer strategy

As described in Section 3.4.2, the number of key hypothesis (a measure of the offline phase of the attack complexity) can be reduced at the cost of the amplitude of the correlation traces using a divide and conquer approach.

The same implementation as in the first case of Section 3.5.2 is considered: the

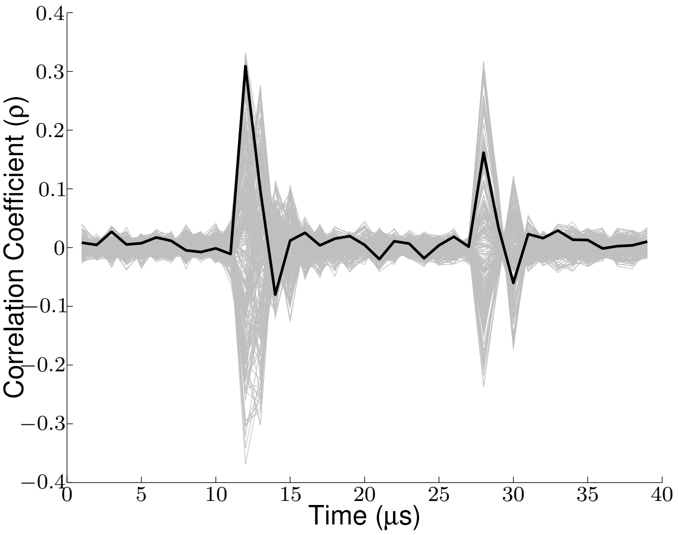


Figure 3.7: Unsuccessful recovery of the 8th byte in a key block

third operation is a modular addition and the device leaks in the Hamming distance power model. The only difference with the latter case is that nibbles in the key words are targeted instead of bytes. According to Equation (3.4), the relation between the correlation coefficients of the two cases is linear and the correlation for nibbles is a factor of $\sqrt{\frac{8}{16}} \approx 0.707$ less than for bytes. This is verified in Figure 3.11, where the correct key can be identified from the power traces.

The second key nibbles yield a little more amplitude in the correlation trace, as shown in Figure 3.12.

3.5.4 One block is known

The same implementation as in the two last sections is considered, with the modular addition as combination operation and the Hamming distance as power model. However, the second of the two blocks is known, thus halving the number of hypothesis. The key was again correctly retrieved. The correlation traces are plotted in Figure 3.13.

3.6 Application to Threefish

We present in this section the results of the experiments of our described methods against the Threefish block cipher. The implementation of the Threefish algorithm for our 8-bit AVR microcontroller is presented below.

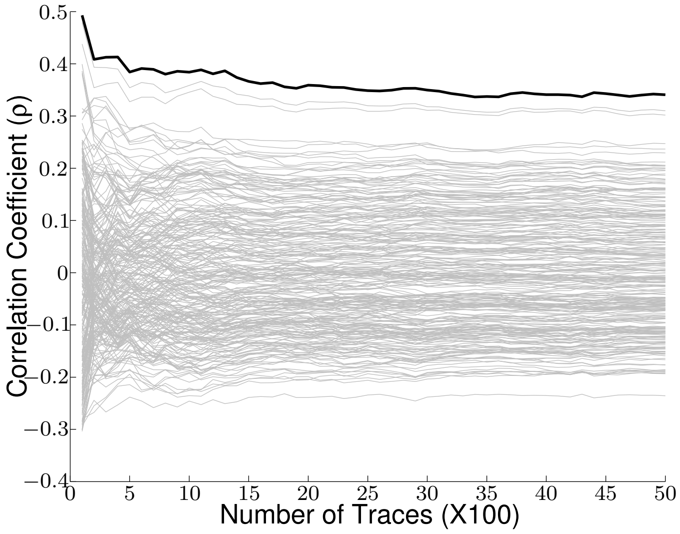


Figure 3.8: Amplitude of the highest correlation peak in relation to the number of traces in the recovery of the 2nd byte

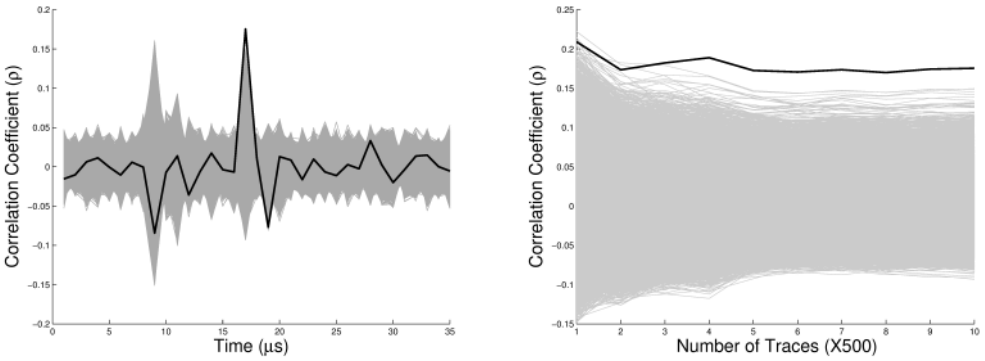
The Fhreefish library

Fhreefish, the cryptographic library for 8-bit AVR platforms we chose for our experiments, is recommended by the designers of Skein on their website for its high efficiency [118]. It is developed by Jörg Walter of *Syntax-K* for Threefish and Skein for the 256-bit block size and is written in assembler.

3.6.1 Experimental results

As represented in Figure 3.1, each of the key and plaintexts are 256-bit long and divided into four 64-bit blocks. The plaintext is loaded into 32 registers. Then, the plaintext blocks are added with the key blocks and stored in the same registers. During the MIX function, the resulting blocks of these additions are pairwise added with each other and the results stored back in the registers. This is the result of this modular addition, which happens inside the MIX function, that we choose to target in our experiments.

As shown by the correlation traces in Figure 3.14a, we can successfully identify the two key nibbles involved in this addition. The correlation peak of the correct key stands out after the analysis of 2500 power traces, as we show in Figure 3.14b. If we target a pair of key *bytes* instead of nibbles, that is, without use of the divide and conquer method described in Section 3.4.2, the correlation coefficient of the correct key byte is larger and stands out more quickly—1500 traces are required—, but the number of pairs of key byte hypothesis jumps from 2^8 to 2^{16}, and the total number of hypothesis considered in the full attack of Threefish-256 increases from $4 \times 8 \times 2 \times 2^{2\times4-1} = 2^{13}$ pairs of nibbles to $4 \times 8 \times 2^{2\times8-1} = 2^{20}$ pairs of bytes.



(a) Correlation traces

(b) Correlation evolution

Figure 3.9: Recovery of a pair of bytes in two key blocks with modular addition as combination

3.7 Conclusion

The subject of this chapter was the application of DPA attacks on the modular addition. This operation, found in a number of block ciphers and hash functions, is inherently a more resistant target function in DPA attacks than the S-box due to its weaker non-linearity and the carry bits which are propagated over the intermediate value.

Conventional DPA attacks fail to recover all the key bits involved in a modular addition because of the carry bit propagation. While Zohner et al. put forward a more sophisticated statistical distinguisher to circumvent the problem in the Hamming weight power model, we propose to target a more complex operation than a simple modular addition, that is, the combination of two additions by a third one or by an XOR. This practical approach has the advantage of being more generic with respect to the power model, but it also has the disadvantage of targeting twice as many bits of the key (except in some cases as for the block cipher TEA and its variants), which quadratically increases the number of key hypothesis. We propose to apply a divide and conquer approach to maintain the attack complexity at a feasible level, at the cost of a factor of amplitude in the correlation traces.

We provide practical evidences of the efficiency of our methods with experiments on simple operations, and verify that they still hold for a high performance implementation of Threefish on an 8-bit AVR microcontroller.

This practical investigation suggests that the choice of a more complex target function may help increase the non-linearity (thus make the correlation coefficient of the correct key more clearly stand out from the other coefficients) and decrease the discrepancies induced by the carry bits in the modular addition.

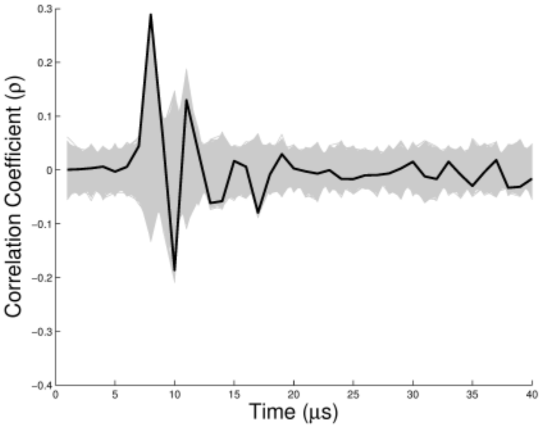


Figure 3.10: Successful recovery of a pair of bytes in two key blocks with XOR as combination

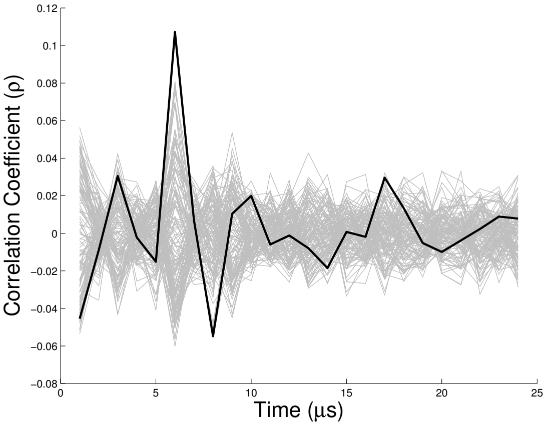


Figure 3.11: Successful recovery of first key nibbles

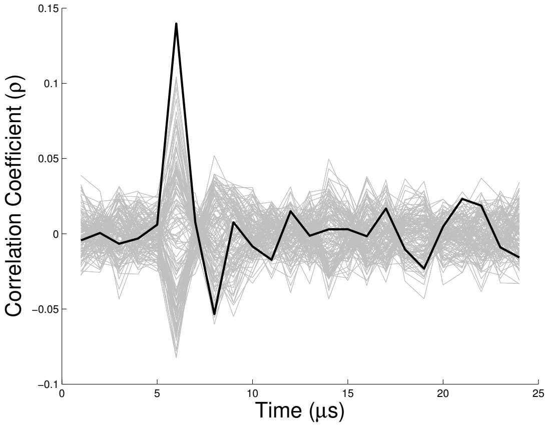


Figure 3.12: Successful recovery of second key nibbles

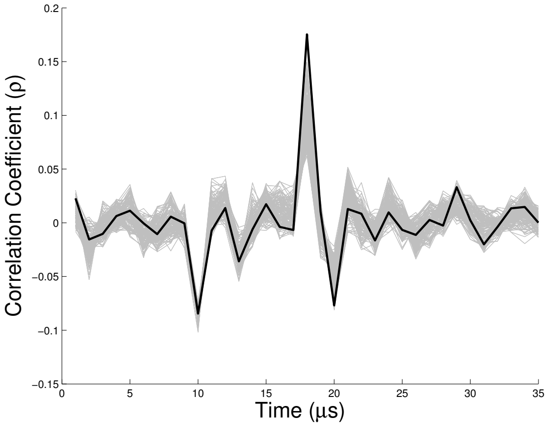
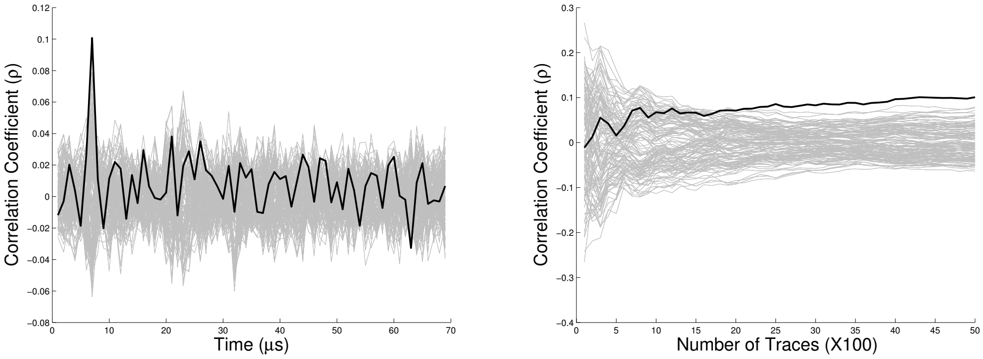


Figure 3.13: Successful recovery of the first byte when one block is known



(a) Correlation traces

(b) Correlation evolution

Figure 3.14: Recovery of the first nibbles in two key bytes in Threefish

Chapter 4

Microarchitectural Trojans

Hardware Trojans are tiny modifications of the design of a chip which may provide a covert communication channel with an adversary, as well as control of the device or disruption of its normal functioning. In this work we study a particular class of hardware Trojans which can be implemented in a chip in order to induce a side-channel leakage or to inject a computational fault during the execution of arbitrary cryptographic software in order to convey secret key material to the attacker. We discuss various software-based activation mechanisms and the different ways in which a fault or side-channel attack exploiting the Trojan effect would allow an adversary to recover secret or private key material. Finally, we describe two practical scenarios respectively in symmetric (against AES) and asymmetric (against RSA) schemes where an adversary would be able to mount an implementation attack with minimal knowledge on the target software based on the vulnerability introduced by the Trojan.

This is a joint work with Johann Großschädl, Neil Hanley, Markus Kasper, Marcel Medwed, Francesco Regazzoni, Jörn-Marc Schmidt, Stefan Tillich, Marcin Wójcik, published in the proceedings of INTRUST 2010 [46]. Part of this work will also be included in the Ph.D. thesis of Markus Kasper and Marcin Wójcik.

Contents

4.1 Generalities

Side-channel leakage is information unintentionally present in the physical characteristics of a device while performing computation. Whereas this leakage weakens the security of a physical cryptographic implementation and therefore is aimed to be minimized by a chip designer, conversely the latter also has the capability to deliberately introduce or amplify, on demand, the side-channel leakage of information meant to be kept secret. Likewise, while a chip designer aims at producing a chip as reliable as possible, she may also want her chip to inject, on demand, a computational fault. These two mechanisms can serve as a backdoor to cryptographic implementations and can be seen as a "constructive" use of implementation attacks. We refer to these two sorts of hardware Trojans as microarchitectural Trojans, because they are nested within the device circuitry. In this chapter, we elaborate on possible designs and specificities of microarchitectural Trojans.

It is important to remark that microarchitectural Trojans can be the initiative not only of a malicious designer, but also that of a malicious manufacturer, as chip fabrication is nowadays outsourced to worldwide third parties. Since the multiplication of actors in the fabrication process from design to system manufacturing implies more threats to the security of a system, the malicious modifications of a hardware design as well as the techniques to detect them have motivated intensive research both from academia and industry [4,44,57,122]. Microarchitectural Trojans can encompass various malicious activities, such as the deactivation of the chip, the production of erroneous results, provide access to the system through a secret backdoor, or the transmission of information through covert channels (this chapter covers microarchitectural Trojans using the side-channel leakage as a covert channel). A Trojan Side-Channel (TSC), as defined by Lin et al. [73, 74], is the modification of a design that uses the physical leakage of a device as a covert channel to convey secret information. A classification of different Trojan types is given by Tehranipoor and Koushanfar [110].

The security of numerous embedded cryptographic applications may be at risk in presence of microarchitectural Trojans, including smart cards, Trusted Platform Modules (TPMs) and other security tokens. A TPM for example serves in Trusted Computing (TC) as a *root of trust*[1] (or *root anchor*). If a microarchitectural Trojan is inserted in a TPM, the entire security and trustworthiness of the system is at risk. However, a TPM-functionality can also be implemented in software and run on a

[1] The Trusted Computing Group (TCG) defines a root of trust as "a component that must always behave in the expected manner, because its misbehaviour cannot be detected" [54]

trojanized processor. As emphasized by Waksman and Sethumadhavan [117], if the processor can not be trusted, no security guarantee can be provided by the system.

Hardware Trojans are by nature very difficult to detect, while possibly being able to cause large damages to the security of a chip or a network: identity fraud and communication eavesdropping and more specifically to programmable chips (FPGAs): IP theft, re-programming or reverse engineering. The different approaches for hardware Trojan detection investigated in the last years fall into three categories. First, the full-reverse engineering of the chip, where by the mean of a thorough analysis of the functionalities the design details are recovered, can be carried out [112]. However, this expensive and time-consuming scrutinizing will not work if the Trojan is present in some of the deployed chips but not in the one under investigation. Second, as already largely performed by the designers themselves after chip fabrication, the chips can be run on a large set of test vectors and their responses be compared with the expected ones [122]. Fuzzing techniques may also be applied. However, the complexity of nowadays architectures renders impossible the exhaustive verification of all the device states. Moreover, testing the outputs will not reveal a Trojan intended to leak information through side-channels. The third approach is to perform a DPA analysis on the test vectors [4, 58] against golden samples. This approach can help detect any abnormalities in a given side-channel which emanates from the device. However, the complexity of such analysis remains highly time-prohibitive and dependent to the precision of the measurement setup. Very recently, Skorobogatov and Woods introduced a new technique for DPA processing called Pipeline Emission Analysis (PEA) [106] to extract the AES key from a secure FPGA chip. Their patented technique allowed them to scan the silicon layer of the chip and find out the existence of an undocumented backdoor for accessing and rewriting the configuration file of the FPGA chip [105].

Throughout this chapter, we term as *attacker* an adversary whose goal is to recover the key or try to make the device behave in an unintended way. Although the attacker may be aware of the presence of the Trojan and know the special activation pattern, from a cryptographic point of view such key recovery is illegitimate.

4.1.1 Contributions

In Section 4.2, we present different implementation attacks which can inspire the design of a microarchitectural Trojan: induced fault attacks, latent fault (or bug) attacks and side-channel attacks exploiting the early-terminating multiplication feature present on some microprocessors. In Section 4.3, we explain the purpose and detail the design possibilities of an activation system in a Trojan. In Section 4.4, we elaborate on the second component of a Trojan: its payload section, of which the goal is to perform the devious action of the Trojan (secret information transmission or fault injection). In Section 4.5, we give the details of two realistic scenarios which suit our Trojans' descriptions. We conclude in Section 4.6.

4.2 Related attacks

In this section we present implementation attacks which can inspire the design of microarchitectural Trojans: induced fault attacks, latent fault (or bug) attacks and microarchitectural side-channel attacks via early-terminating multiplications. The purpose of a microarchitectural Trojan is to create (or amplify) the applicability of these attacks.

4.2.1 Fault attacks

Fault attacks are implementation attacks, where the normal functioning of a device is disturbed by an intentional *fault injection* in the first stage of the attack. In a second—offline—stage, the effect of the malfunction is analysed. In most cases, the faulty output is considered, if necessary in comparison with a correct output obtained from the same input. Fault attacks are *active* implementation attacks, because the computation flow is disrupted by the attacker, as opposed to side-channel attacks, which are *passive* implementation attacks because the monitoring of the device is made during normal functioning.

There exist many ways of disturbing the normal behaviour of a device. *Non-invasive* ways, i.e. which do not require a physical alteration of the device, include the sudden change of voltage in the power supply line (called a *spike*) or of the clock frequency (referred to as a *glitch*). Their insertion during a computation can affect the data transmitted over the bus, and usually leads to data corruption or skipping of instruction [7].

Optical fault injection require the decapsulation of the chip and thus are referred to as *invasive* attacks. At a particular space location, the injection of a light flash or a laser beam can also disturb the computation process. The advantage of the laser is the capability of accurately positioning the fault injection, compared to light flashes or spikes and glitches [97].

The fault type (spike, glitch, light flash, laser beam), the time and location of the fault injection and, if predictable, the fault effect on the computation (bit flips on certain data, instruction skip) define a *fault model.* A fault model can induce a key recovery from the exploitation of the faulty output. For an attack to work in practice, the choice of a realistic fault model is crucial.

On secret-key schemes, fault attacks can be differential, that is, the differences between the correct and the faulty outputs are analysed [16]. Fault attacks on secret-key schemes can also be non-differential, skipping individual rounds (thus reducing their number and decreasing the complexity of the cipher) [33] or deleting the S-box table [98].

On public-key schemes, fault attacks do not require the result of a correct encryption. In the well-known example of the Bellcore attack [22] (described later in Section 4.2.2), only one faulty output is required to retrieve the factorization of the RSA public modulus. Its simplicity makes the Bellcore attack relevant for the design of a microarchitectural Trojan inducing a fault.

4.2.2 Bug attacks

In 2008, Biham et al. introduced a new type of attack against public- and secret-key schemes, the bug attacks, which exploit an accidental erroneous result of a microprocessor occurring while it multiplies a specific pair of operands [15]. Such unexpected behaviour notoriously happened with the Pentium processor and its division bug in the mid 1990's. With the increasing complexity of modern microprocessors, the functional testing which aims to detect such bugs is made more difficult, and it is likely that more bugs will slip through. In this context, the insertion of microarchitectural Trojans is also likely to remain more easily undetected.

Biham et al. present different attacks carried out on a processor which outputs an erroneous result for the multiplication of the two words a and b, which are assumed to be known to the attacker.

The first attack presented, targeting RSA-CRT, retrieves the factorisation of the public modulus following the steps of the Bellcore attack [22], and is of great interest for the design of a Trojan injecting a fault, because it requires only one ciphertext to retrieve the factorization of the RSA public modulus. We outline its steps.

The Bellcore attack

The RSA modulus is $N = pq$ where the secret factors p and q are two large primes, and we state without loss of generality that $p < q$. The pair of secret and public keys used by the target processor is (d, e).

First, a ciphertext C is selected by the attacker such that: (1) it is comprised between p and q (without knowledge of the secret factors, i.e. it has to be close to $\sqrt{N}$); and (2), the squaring of C occurring during the decryption routine involves the multiplication of a by b, known to the attacker, resulting in an incorrect decrypted message $\tilde{M}$. We describe later in Section 4.3.2 how to select such ciphertext.

Second, the ciphertext is submitted to decryption on the target buggy microprocessor. In RSA-CRT, the deciphered message M is obtained by the mean of two exponentiations to the power of d modulo p and q. The first step of decryption is to reduce C modulo the secret factors.

$$\begin{aligned} C_p &= C \bmod p \\ C_q &= C \bmod q = C \quad \text{(remember that } C < q) \end{aligned}$$

In the second step of an RSA decryption, as part of the square-and-multiply algorithm used for exponentiation, C_p and C_q are squared, possibly multiplied if the corresponding bit of the exponent is set, and so on iteratively, for all the bits of the exponent. The values M_p and M_q are obtained.

$$\begin{aligned} M_p &= C_p^{d_p} \bmod p \\ M_q &= C_q^{d_q} \bmod q = C^{d_q} \bmod q \end{aligned}$$

where $d_p = d \bmod p - 1$ and $d_q = d \bmod q - 1$. Two integers μ and ν, such that:

$$\begin{cases} \mu \equiv 0 \pmod{p} \\ \mu \equiv 1 \pmod{q} \end{cases} \quad \text{and} \quad \begin{cases} \nu \equiv 1 \pmod{p} \\ \nu \equiv 0 \pmod{q} \end{cases}$$

exist and are easy to compute. Because μ and ν are coprime, the decrypted message can be reconstructed as follows:

$$M = \mu M_p + \nu M_q \pmod N$$

The use of the Chinese Remainder Theorem speeds up by a factor of 4 the RSA decryption and signature algorithms, which explains its prevalence in RSA implementations.

Now, recall that the decryption is performed on a processor that *erroneously* compute the multiplication of a by b, and that C is such that the squaring of C involves this multiplication. Therefore,

$$C^d \bmod q = \tilde{M_q} \neq M_q$$

thus

$$\tilde{M} = \mu M_p + \nu \tilde{M_q} \pmod N \neq M.$$

However,

$$M - \tilde{M} \equiv \nu(M_q - \tilde{M_q}) \pmod N,$$

hence, since $\nu \mid q$, $\nu \nmid p$ and (with very high probability) $p \nmid M - \tilde{M}$, the secret factor q can be retrieved by computing:

$$q = \gcd(M - \tilde{M}, N)$$

From this equality, we note that $M \equiv \tilde{M} \pmod q$, and thus: $C = M^e \equiv \tilde{M}^e \pmod q$. Therefore, the knowledge of a correctly decrypted ciphertext M is not required, as it was shown by Lenstra [71], which is particularly interesting in our context:

$$q = \gcd(C - \tilde{M}^e, N)$$

Other attacks exploiting a computational bug

Biham et al. present other attacks with higher time complexity and number of required ciphertexts: against the Pohlig-Hellman cipher [87], against RSA decryption with and without use of the Chinese Remainder Theorem (CRT) and RSA decryption along with OAEP [10]. These attacks assume left to right (LTOR) exponentiations, i.e. running from the MSBi to the LSBi of the secret exponent. On right to left (RTOL) exponentiations, i.e. the opposite case, attacks are presented also targeting Pohlig-Hellman and RSA decryption.

In some sense, since a computational bug is a sort of hidden feature of a processor which in some circumstances is capable of leaking secret information, it can be seen as an *accidental* microarchitectural Trojan.

4.2.3 Early-terminating multiplications

On a high number of 32-bit processors, a (32×32)-bit multiplication $a \times b$ is executed in an iterative manner, processing the multiplicand a with 8 bits of the multiplier b

at a time. The intermediate product is then shifted and added to the intermediate result. When the remaining bytes of b are zero, the multiplication terminates "early" and the 64-bit result is written back to the registers. Most applications, in particular in the field of Digital Signal Processing (DSP) and multimedia, heavily rely on multiplication operations, and the *early termination* of the `mul` and `umul` operations significantly improves the performance of the processor.

On the ARM7TDMI processor, as an intermediate multiplication takes *one* clock cycle, a full multiplication $a \times b$ takes from 2 to 5 clock cycles to complete, depending on the multiplier b, including one clock cycle of constant latency [72]. When (1) the three most significant bytes (MSBys) of b are zero, (2) the two MSBy of b are zero and its third MSBy is non-zero, (3) the MSBy of b is zero and its second MSBy is non-zero, or (4) the MSBy of b is non-zero, the multiplication is executed in respectively $2, 3, 4$ and 5 clock cycles. The ARM7TDMI has an instruction pipeline composed of the following stages: Fetch, Decode and Execute, taking each one clock cycle. When a multiplication instruction takes from 2 to 5 clock cycles, the pipeline is stalled during 1 to 4 clock cycles because the Fetch and Decode stages are waiting for the Execute stage to complete, inducing a decreasing power consumption of the chip. The pipeline stalls are clearly observable on power measurements, and thus the different early-terminations clearly distinguishable, as shown in Figure 4.1.

This induced microarchitectural side-channel is the security cost of the performance improvement provided by the early terminations when running cryptographic software. Großschädl et al. have shown that the different latencies depending on the multiplier b allow an attacker to deduce information in a "coarse-grained" way through the timing of the multiplication, which eventually allows the attack to be mounted remotely, even over the Internet. The leakage is also observable in a "fine-grained" way through the power consumption, which is suitable for embedded devices [53]. Like the ARM7TDMI, other processors are also subject to an early-terminating multiplication attack, such as the StrongARM SA-1100, the MIPS32 4Km and certain PowerPC models.

In some sense, the early-termination mechanism can be seen as an *unintentional* microarchitectural Trojan.

4.3 Activation mechanisms

We suggest in this section how a microarchitectural Trojan can be activated so as to perform its malicious activity with minimal chance of being discovered. Additionally, we explain how and why to input a *parameter* along with the activation pattern.

Following the Trojan taxonomy suggested by Wolff et al. [122], a Trojan divides into two parts: the activation mechanism, also called trigger, and the payload section. To remain undetected in particular during the functional testing of the chip, a microarchitectural Trojan features a mechanism that triggers its functionalities. The trigger of a hardware Trojan can be subject to external or internal conditions [110]. A Trojan is externally triggered in response of a signal received through an antenna or some sensor. A Trojan is internally triggered when the activation conditions happen inside the chip. These internal conditions can be for example pre-defined

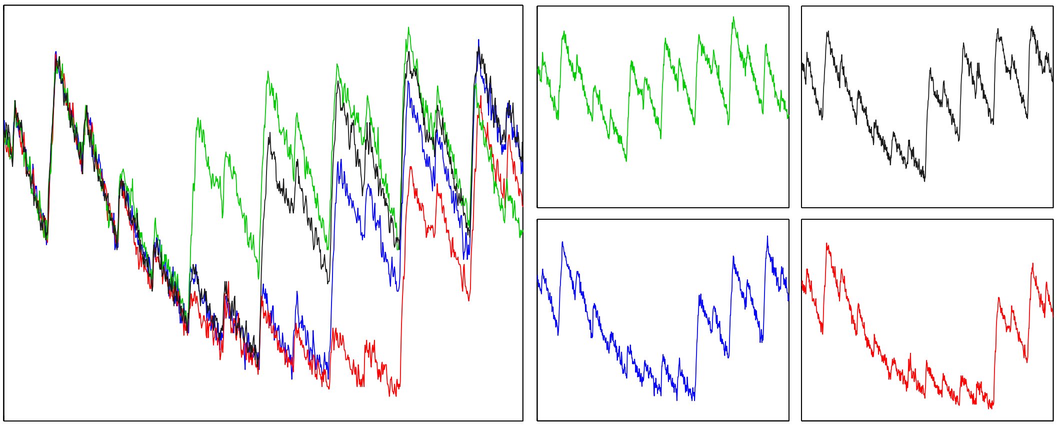


Figure 4.1: Overlaid (left) and individual (right) power consumption traces showing ARM7 multiplications that take 2, 3, 4 and 5 clock cycles (top left to bottom right) [46].

input values, a time, clock cycle or execution count or the sequence of some special logic values. As internal activation does not require any peripheral as opposed to external activation, the former can be implemented with only a set of combinatorial gates and will take only a very low area and performance overhead. In the rest of the chapter we focus on internal activation mechanisms, for their greater relevance in the design of a hidden functionality.

An internal activation should be carefully chosen so that the Trojan is not enabled during the functional testing of the chip, nor is it enabled by mistake by the user. In the following, we present two methods for an internal activation of the Trojan. In the first one the chip snoops the data bus waiting for a pre-defined input value while in the second the operands of a certain instruction are snooped. In both cases, a pre-defined value initiates the activity of the Trojan. This value should be long enough and possibly authorize the passing of a parameter which for example allows an attacker to define during which execution the payload section should occur. We show later why it is necessary, especially against block cipher implementations.

4.3.1 Method 1: Snooping the data bus

In this first activation method, a comparison module (which is part of the Trojan) lies between the CPU and the data cache, snooping all data transmitted over the bus. This module simply compares the data blocks against a pre-defined value, which can be hard-wired (and thus cannot be changed) for a minimal area overhead and insignificant latency.

Because of the application to AES we present in Section 4.5, we suggest the activation pattern to be of length 16 bytes, possibly divided into 12 bytes for the activation of the Trojan and 4 bytes for its parametrization. A random sequence of 12 bytes has an entropy of 96 bits and is quite unlikely to appear in the ASCII characters which are intended to be passed on over the data bus. Accidental activation can be even more cautiously obviated if a counter is set, that requires the activation pattern to be input a certain number of times before triggering the behaviour of the Trojan.

We suggest that the same pattern is used for both activation and deactivation so as to keep the area overhead minimal.

The activation of the Trojan can eventually take place within a challenge-response authentication protocol, in which two parts interact: a verifier and a prover. The former wants the latter to prove that she is in possession of a secret key without publicly disclosing it. For this purpose, the verifier sends the prover a nonce, the prover receives it in its register file, encrypts it under the secret key and sends it back to the verifier. Let us assume that the prover's AES code runs on a trojanized processor. Then, an attacker can play the role of a verifier and sends the activation pattern as a nonce, several times if specified, and monitor the side-channel leakage induced or amplified by the Trojan.

4.3.2 Method 2: Snooping operands of instructions

Public-key cryptosystems involve the execution of numerous long-integer arithmetic operations, like a modular multiplication, such as in an RSA exponentiation or in a point multiplication on an elliptic curve defined over a large prime field. Besides its prevalence in asymmetric cryptography, the modular multiplication is also found in many block ciphers, such as RC6 [94], IDEA [69], DFC [50], MARS [29], MultiSwap [101] and Nimbus [75], in the stream cipher Rabbit [18] and in the message authentication code UMAC [17]. Also, the `mul` instruction is used in the optimized software implementation of the AES by Gladman [52] to speed up the execution of the `MixColumns` transformation.

In this second method for the Trojan activation, the operands of a specific instruction, here at the example of the `mul` instruction, are compared against predefined values. The `mul` instruction takes as input two integers a and b, each of length 32 or 64 bits depending on the size of the registers. Architectures with smaller sizes are not considered since a full functional testing of the processor would then reach a feasible complexity. On a 32-bit processor, the chance of accidentally activating the Trojan would be $2^{-32-32} = 2^{-64}$. We observe that in case of the squaring of a long integer composed of the words a and b, both multiplications $a \times b$ and $b \times a$ are involved. Hence, the chance of accidental activation is almost doubled, yet remaining sufficiently low: less than 2^{-63} and 2^{-127} on 32- and 64-bit architectures respectively.

A scenario where an attacker has the ability of feeding the processor with a pair of operands of his choice so as to activate the Trojan is described by Biham et al. [15]. We explain here how a ciphertext can be chosen so as to involve a desired multiplication $a \times b$ which in our scenario triggers the Trojan. Let us assume a CRT-based RSA decryption running on a trojanized processor. P and Q are the secret factors (unknown by the attacker) of the (known) modulus N, and we state without loss of generality that $P < Q$. The integer $\lfloor\sqrt{N}\rfloor$ is comprised between P and Q, and any integer close to $\lfloor\sqrt{N}\rfloor$ satisfies this condition. The attacker replaces the two least significant words of $\lfloor\sqrt{N}\rfloor$ by a and b to form a “poisonous” ciphertext C, which she submits to the trojanized processor. Upon fectching C in its registers, the multiplier reduces C modulo P and Q. Note that $C \bmod Q = C$ because $C < Q$. Then, the reduced ciphertexts are squared modulo P and Q respectively. The squaring of C modulo Q involves the multiplications of a by b and b by a, and one of which fires the trigger. To summarize, the adversary can activate the Trojan by feeding the decryption function a carefully manipulated ciphertext.

4.4 Effects of the Trojan

Following the description of the possible activation methods, we now elaborate on the second component of a Trojan: the payload section. Once activated, the Trojan can be designed to perform various devious actions, which divide into two sorts: the injection of one or several computational faults and the insertion of a side-channel leakage. Whereas the first kind makes the cryptographic routine output an erroneous result, the second leaves the output correct but requires an access for an attacker to the observation of a side-channel leakage: execution time, power consumption or electromagnetic radiations.

The conveyance of secret information is carried out by one trojanized instruction, but its malicious action should not occur every time the instruction is called once the Trojan is activated. In order to keep the Trojan hidden, especially when it inserts a fault, its devious activity should occur only at the points dictated by the attacker. We count three possible action scenarios.

- Only one call to the trojanized instruction is corrupted upon activation. This can be the same call as the one which triggered the Trojan, or the instruction can wait for a specific count of this instruction or a certain number of clock cycles before enabling the payload section. All other executions of the trojanized instruction are performed normally. We note that a trojanized `mul` instruction which, upon reception of the pre-defined activation operands a and b in its two input registers, flips a bit of the product $a \times b$ in the output register, is similar to the setup of a bug attack [15].

- A certain number of calls to the trojanized instruction are affected upon activation. This number of leaking instructions is either pre-defined or determined by the attacker through the parametrization of the Trojan, possibly supported in the activation phase. The first artificially leaking instruction can occur after a certain number of executions or clock cycles, again either pre-defined or requested by the attacker through parametrization. In Section 4.4.1, we show how such a Trojan operating in 16 consecutive calls to a `xor` or a `load` instruction can recover the key in an AES implementation.

- The Trojan operates as long as it is activated until deactivation. We recall that deactivation is processed in the same manner as activation, using the same pre-defined input. Every call to the trojanized function artificially leaks information, hence this Trojan is suitable in the context of timing and power analysis attacks, as we show in Sections 4.4.2 and 4.4.3.

We now give detailed examples on how a trojanized instruction can disclose secret information through the insertion of a computational fault in Section 4.4.1 and through the induction of a side-channel, in Sections 4.4.2 and 4.4.3.

4.4.1 Fault induction

Fault inductions is the simplest way in which a microarchitectural Trojan can leak information. In the following, we present two designs of fault induction that a Trojan can follow to recover the key. In a first one, suitable against block cipher implementations, the lookup table returns zero instead of the actual entry value. In the second design, suitable against RSA implementations, a trojanized instruction flips one bit of the result.

Zero lookups

Let us consider a software implementation of the AES that uses a lookup table for the S-box. In the following example, the microarchitectural Trojan is embedded in the `load` instruction, but a similar attack could be devised with the `xor` instruction. Here, the `load` instruction is designed to return zero for the 16 bytes of the final `SubByte` function (the cipher structure is outlined in Section 2.2.5). Note that setting a function to a known value during the execution of a block cipher is equivalent to reducing its number of rounds to the number of rounds performed after these zero lookups. In our example, after the final `SubByte` function, the remaining operations are `ShiftRows` and `AddRoundKey`. Therefore, if the final `SubByte` function returns zero, the final round key is directly output as the ciphertext, allowing the attacker to deduce the master key with a reverse application of `KeySchedule`. Unlike the usual fault attacks against the AES, namely differential fault attacks, no correct encryption is needed in this setup.

Meanwhile, in a practical implementation of this zero lookup attack, an adversary would have to deal with two main issues. First, the activation of the Trojan takes place at the beginning of the encryption, upon matching of the plaintext value with a hard-wired pre-defined activation pattern. But as explained in Section 4.2.1, the fault has to be inserted in the final round of encryption. The implementation details may be unknown by the attacker, hence the count of `load` instructions for which the activation of the Trojan has to be delayed after triggering may also be unknown. This issue can be overcome with the passing of a parameter along with the activation pattern, as described in Section 4.3.1, and some “trial and error” session, unless the attacker knows the details of the AES software running on the target processor.

A second issue an attacker would have to deal with in practice are context switches, which occur in multitasking environments and where different processes interleave with each other. With context switches, the count of `load` or `xor` executions becomes more difficult to predict.

Single bit-flip in instruction execution

A trojanized instruction can be designed so as to flip one or several bits of the instruction output upon activation. In this case, the injected fault meets the models of differential fault attacks on symmetric ciphers [16, 39, 114], and as well that of RSA in the bug attacks [15].

Against AES implementations, the `xor` instruction is a good choice of function to trojanize because it directly manipulates the round key in the `AddRoundKey` function. Tunstall et al. [114] presented a differential fault attack on AES that allows an adversary to reduce the key entropy from 128 down to 8 bits (i.e. 256 key candidates) with a single pair of correct and faulty ciphertexts, assuming that a random byte fault is injected at a known position of the state in the 8th round. Like in Section 4.4.1, the Trojan inserts a fault several executions of `xor` instructions after recognition of the activation sequence, thus the attacker should be able to define the delay length through parametrization.

Against implementations of RSA [95] and that of other public-key cryptosystems, the natural candidate for implementing the trigger mechanism is the `mul` instruction, where two pre-defined 32-bit operands can activate the Trojan in an RSA decryption. Section 4.3.2 details how to construct such activation ciphertext. Unlike Trojans targeting block cipher implementations, a Trojan against RSA implementations can use the `mul` instruction for the activation and the payload, i.e. the bit flip in the result of a multiplication. This scenario would meet the conditions described by Biham et al. for a CRT-based implementation of RSA [15] and allow an attacker to retrieve the key used in a decryption operation with a single chosen ciphertext. The key recovery follows the steps of the Bellcore attack [22] that we outlined in Section 4.2.2.

Against implementations of block ciphers which use multiplications, such as IDEA [69], RC6 [94] and DFC [50], the injection of a fault in the `mul` instruction may also allow an attacker to gain information on the round key multiplied with a known intermediate value.

4.4.2 Timing variation

Besides the injection of a fault during the computation, a Trojan can also be designed to make the latency vary depending on the key bytes being manipulated. Otherwise stated, it can introduce an artificial side-channel, observable through the timing of execution. The monitoring of the execution time does not require a physical access to the device and can even be performed over a network, including the Internet. A microarchitectural Trojan can introduce a delay in essentially two ways: (1) it can stall the pipeline for a certain number of clock cycles; or (2) it can flush the cache memory, for a less predictable number of clock cycles of delay. Nevertheless, both methods significantly increase the dependency between the key and the execution time, in addition to the possibly inherent timing side-channel induced by the implementation. Therefore, a chosen plaintext strategy like the timing-driven cache-based attack of Bernstein [12] can be applied. In the following, we explain how a microarchitectural Trojan can reveal information about the secret key used in software implementations of the AES and RSA via a manipulation of the latency of the `xor` and `mul` instructions respectively. As previously, we consider a 32-bit architecture.

AES and other secret-key schemes

Recall that in the AES, the plaintext is first added with the key, by the use of the `xor` operator. Against AES implementations, the `xor` instruction is thus a natural candidate for a Trojan implementation. Among the 4 bytes output by the `xor` call, if the least significant byte (LSBy) is zero, that is, if the LSBys of the 32-bit key and plaintext words are equal, the Trojan stalls the pipeline by one clock cycle. Analogously, the pipeline is stalled by 2 and 4 clock cycles if the second and third output LSBys are equal to zero; and finally, the pipeline is stalled by 8 clock cycles if the MSBy of the output is equal to zero. The number of clock cycles taken by the execution of this `xor` call ranges from 1 to 16, and each amount of delay determines which of the four output bytes are zero. For each zero output byte, a key byte is identified. Consequently, once the Trojan activated, the attacker can submit plaintexts of his choice to identify the key bytes input in an AES `AddRoundKey` function. The number of clock cycles 1, 2, 4 and 8 for which the pipeline is stalled are given as examples, but note that any superincreasing sequence, i.e. where each term is greater or equal to the sum of the previous terms, fulfils the requirement, namely: an adversary is able to identify the zero bytes output by the `xor` instruction from the variations of latency.

For each 4-byte word, about 204 submissions are required on average, according to simulations. Therefore, since the recoveries of the four 4-byte words are independent, about 816 submissions to the trojanized device are required. Parametrization, here, can help the attacker target only one `AddRoundKey` function and recover only one 4-byte key word at a time.

RSA and other public-key schemes

Against RSA and other public-key schemes implementations, a similar attack can be mounted when the Trojan increases the latency of the `mul` instruction depending on the magnitude of one of the two operands. The Trojan stalls the pipeline when the LSBy of the 32-bit multiplicand is zero (and the three other bytes are non-zero), and similarly, the Trojan stalls the pipeline for 2 and 4 cycles when the second and respectively third LSBys of the multiplicand are zero (other bytes being non-zero). And finally, when the MSBy is the only byte of the multiplicand being zero, the Trojan stalls the pipeline for 8 clock cycles. The number of clock cycles of the additional latency of the `mul` instruction ranges from 0 when none of the multiplicand bytes are zero, to 15, when the entire multiplicand is zero. This can be seen as the inverse of the early-termination effect presented in Section 4.2.3 and may be referred to as *late*-termination effect. This artificial side-channel can help an adversary deduce the sequence of modular multiplications and modular squaring operations which compose an exponentiation and, if the exponent is secret, allow an adversary to recover its bits. Note that this approach works for “textbook” RSA as well as for optimized implementations, including when the plaintext or ciphertext is padded according to PKCS #1 [53, 59].

4.4.3 Power variation

If an attacker is able to monitor the power consumption taken by the trojanized microprocessor during an artificially leaking instruction as the `xor` and `mul` described in the previous section, she will directly observe the different latencies induced by the Trojan. On power traces, not only the overall execution time is measurable, but the periods of artificial inactivity induced by the Trojan are distinguishable from the periods of activity of the microprocessor. Indeed, at the circuit level, the power consumption of a gate is composed of a *static* component (which is constant and typically low) and a *dynamic* component (which is relatively much higher). The latter is determined by the switchings occuring at the outputs of the gates [76]. When little switching activity occurs (due to the pipeline stall), the power consumption of the processor drops down to its static part, as an attacker would easily observe using a typical measurement setup for power analysis. Therefore, the information leaked via the power consumption of the `xor` and `mul` instructions is the different latencies of individual executions of these instructions, which an attacker can use to mount the attacks described in the previous section.

Note that a microarchitectural Trojan may also be used in a “constructive way” for *digital watermarking* of a microprocessor core [9]. Like in our example of Trojan for the `mul` instruction, an integer multiplier can be designed such that for one particular pair of operands, the pipeline is stalled for a number of clock cycles. Such a pair of numbers is hard to guess on microarchitectures of more than 16 bits, and thus allows a processor designer to reliably identify his IP using a typical measurement setup. The IP owner (or any other entity knowing the correct operand pair) just has to observe the power profile and if necessary compare the execution time taken by the integer multiplication of the pre-defined pair of operands with any other pair of operands. Identifying an IP block through side-channel observations with the help of a digital watermark is considerably cheaper than a full reverse engineering.

4.5 Case studies

In this section, we put forward two realistic scenarios where a suitably designed and implemented Trojan in a general-purpose microcontroller can allow an attacker to recover the AES secret key of a server on the one hand, and the RSA private key of a server on the other hand.

4.5.1 AES

In the following, an attack is mounted against an AES implementation running on a trojanized processor. We use a simple AES-based challenge-response authentication protocol (Figure 4.2), where one entity, the *verifier* sends a nonce, or *challenge*, to the other entity, the *prover*. The prover receives the challenge, encrypts it under the secret key and sends the result back to the verifier, thereby proving that she is in possession of the secret key. In our context, the prover is running on a trojanized processor and the verifier plays the role of an attacker, aware of the Trojan's presence and the activation sequence. The attacker's goal is to recover the secret AES key used by the server. The attacker first sends a nonce containing the pre-defined activation pattern to the prover. At some point in time, the prover loads the nonce into the registers, which activates the Trojan, as described in Section 4.3.1.

$$\text{Verifier (attacker)} \quad \xrightarrow{\qquad nonce \qquad} \quad \text{Prover (trojanized)}$$

$$\xleftarrow{\ \text{AES}_K(nonce)\ }$$

Figure 4.2: Challenge-response protocol, in a Trojan-based attack

Once activated, the Trojan induces variations in the latency of the `xor` instruction, as explained in Section 4.4.2, which can be measured and compared via timing measurements, or directly observed, via power measurements. We assume that the attacker is able to monitor the power consumption of the prover's device. In order to recover a 4-uple of key bytes (referred to as a key word), the attacker sends up to 256 different nonces to the target device, each word containing the same 4 bytes. For each query and for each byte, she aims at finding out the correct key byte, that is, the one that makes the pipeline stall. The number of clock cycles for which the pipeline is stalled during the target `xor` instruction, ranging from 0 to 15, directly tells the attacker which of the nonce bytes were equal to the corresponding key bytes. Within an average number of 204 queries, she will have recovered the 4 bytes of the key word. Then, she moves on to the next key word. An average number of 816 nonce submissions is required for the attacker to retrieve the entire key.

4.5.2 RSA

In the following, we assume an SSL (or TLS) server running on a trojanized processor. The SSL software is using a CRT-based implementation of RSA to perform operations involving the secret key, i.e. decryption and signature generation [95]. In our example, the attacker plays the role of a client wishing to establish a secure connection with the SSL or TLS server. His goal is to obtain the secret key used by the server for key establishment. The possession of this secret key would allow an attacker to decrypt all encrypted communications between the server and other clients, and to impersonate the server while communicating with its clients. In essence, the SSL protocol is composed of two sub-protocols, one of which is the handshake protocol for authentication and key establishment. When using an RSA-based cipher suite, the secret key shared between a client and a server is established via key transport: The client generates a random number, encrypts it under the server's public RSA key, and sends the result to the server. Then, the server performs an RSA decryption and obtains the shared secret.

$$\begin{tikzcd}[column sep=huge]
\text{Client (attacker)} \arrow[r, shift left=2, "C = r^e(N)"] \arrow[r, leftarrow, shift right=2, "\tilde{M} = \text{RSA-decrypt}_d(C)"'] & \text{Server (trojanized)}
\end{tikzcd}$$

Figure 4.3: Key establishment, in a Trojan-based attack.

In order to activate the Trojan, the attacker sends a “manipulated” ciphertext to the server, instead of an encrypted random number. This manipulated ciphertext has to be constructed according to the description in Section 4.3.2. In short, the manipulated ciphertext is an integer C corresponding to the square root of the public modulus N (rounded to the nearest integer), in which the two least significant 32-bit words are replaced by the two operands a and b that trigger the Trojan when a `mul` instruction is executed on them. Once activated, depending on its design the Trojan can inject a fault in the result of the `mul` instruction, in which case the attack procedure is similar to Biham et al.'s bug attack [15]. On the other hand, the Trojan can cause variations in the latency and power consumption of the `mul` instruction. Hence, with an attack like the side-channel key recovery via early-terminating multiplications [53], the key can be successfully recovered.

4.6 Conclusions

In this chapter, we have introduced microarchitectural Trojans, a new class of hardware Trojans specifically designed to induce (or amplify) side-channel leakage emanated by a general-purpose microprocessor when running cryptographic software. We have defined these Trojans along with their “malicious” or “constructive” purpose (interpretations may vary); we have investigated the various possibilities for a chip designer or manufacturer to insert them in the chip, either during the chip design or during the chip fabrication. We have also proposed novel software-based activation mechanisms, and we have elaborated on various methods for a Trojan to reveal the secret key: either by inserting a computational fault (ranging from a single bit flip to a zero lookup), either by increasing side-channel leakages (execution time or power consumption) by the mean of pipeline stalls. We have described two “real world” scenarios where an attacker aware of the Trojan's presence and the activation pattern is able to recover the AES secret key and RSA private key used by an OpenSSL software.

The strength of microarchitectural Trojans is their capability to attack arbitrary cryptographic software (AES and RSA among others), even in presence of sophisticated countermeasures against SPA and DPA attacks that would normally defeat these attacks on genuine microprocessors. Microarchitectural Trojans are also highly difficult to detect through functional testing especially because of the activation mechanism, and through reverse engineering because of their small area overhead.

Chapter 5

Key enumeration in side-channel attacks

In a side-channel divide and conquer attack such as DPA, the key is recovered chunk by chunk in a *divide* stage, and the chunk candidates are combined to form full key candidates, which are tested against a valid pair of plaintext and ciphertext, in a *conquer* stage. Although it has not drawn a lot of attention from the side-channel community, the complexity of this latter stage can quickly reach an infeasible range when the top chunk candidates are not always output first and thus render the recovery of the entire key impossible. In this chapter we address this so-called key enumeration problem and propose to build a full key Probability Mass Function (PMF) out of the individual chunk PMF using pairwise multiplications and a recursive decomposition of the problem. We show the advantage of this enumeration over a straightforward approach. Additionally, we show that it allows an adversary to achieve comparable or better success with lower measurement complexity in the building phase of a template-based DPA attack.

This is an unpublished joint work with Ilya Kizhvatov and Andrey Bogdanov.

Contents

5.1 Introduction

The common target of a side-channel attack is the recovery of the secret key stored within an embedded device implementing a cryptographic algorithm. The most practical key recovery side-channel attacks—DPA and template attacks—are divide and conquer attacks. In these attacks, the physical leakage of an implementation enables an adversary to build a distinguisher for the individual small chunks of the secret key. The individually recovered chunks are then combined to get the full key.

5.1.1 Motivation

The majority of work published on side-channel attacks focuses on improving the physical and algorithmic part of the attack, that is, the *divide* part of the attack, which aims at recovering a single chunk of the key. It is assumed that one can either a) have enough measurements to obtain a single candidate for each chunk or b) if there are several candidates for some of the chunks, run an exhaustive search among all the possible combinations of the chunk candidates. In a realistic situation where noisy measurement environment is combined with a small allowed measurement count, the resulting number of full key candidates can be very large, such that the exhaustive search can be practically infeasible. The problem of how to deal with multiple full key candidates to reduce the overall attack complexity was typically overlooked.

5.1.2 Our contribution

In this chapter, we present an algorithm for obtaining a sorted list of full key candidates using pairwise computations, as we describe in Section 5.5. Our algorithm significantly improves the overall efficiency of DPA. We use a template-based DPA attack to get information about the distribution of the position of the correct guess in the sorted lists for the individual key chunks, because these attacks in particular allow an adversary to derive a PMF of this distribution. From this chunk PMF, the algorithm outputs the PMF of the full key in correspondence with a list of full key candidates sorted by their likelihood. We show the advantage of our algorithm in practice at the example of template-based DPA attacks in comparison with a lexicographical sorting. We demonstrate that an optimized sorting enables an attacker to reduce the profiling stage complexity while keeping the same online attack complexity.

5.2 Related work

Junod and Vaudenay devised an optimal sorting method [61] for Matsui's linear cryptanalysis [77] with lists of small cardinality.

The key enumeration problem in a side-channel attack was first addressed in the work of Dichtl [37] for the particular case of an ad-hoc template attack against an embedded software implementation of DES [41]. The algorithm exploited the probabilities of individual bits to be correct (previously known from a profiling stage) and resulted in an optimal list of full key candidates. It is however not applicable to standard DPA or template-based attacks because information on the key was recovered bitwise.

More recently, Standaert et al. [116] explained how an optimal key enumeration algorithm developed by Pan et al. [86] having a high memory and time complexity can be made applicable through a recursive decomposition of the problem. In the present work, which was done prior to the publication of Standaert et al., the same recursive decomposition of the problem is applied, however our method is not as efficient and requires more computation. We describe later their sorting method in Section 5.7.

5.3 Complexity of divide and conquer attacks

In this section, we present the measures that we use to compare the different enumeration methods.

In the case of AES-128, the target of a full DPA attack is the 16-byte key: $K = (k_0, k_1, \ldots, k_{15})$. The result of the "divide" part of DPA are 16 sorted lists of 256 bytes each, in correspondance with the 256 probability values ζ_j of the PMF. The result of the "conquer" part is a list of 2^{128} full key candidates sorted with respect to their probabilities:

$$Z(i_0, i_1, \ldots, i_{15}) = \prod_{j=1}^{16} \zeta_j^{(i_j)},$$

where i_j is the rank of the correct subkey in the j-th 256-byte list as obtained by a DPA adversary and ζ_j is the probability for the correct j-th byte of the key to appear at rank i_j in the 256-byte list. Examples of subkey PMFs we obtained in our experiments are plotted in Figure 5.1.

A complete enumeration would output 2^{128} candidates sorted by descending values of probability $Z(i_0, i_1, \ldots, i_{15})$, which is the probability that in each subkey list, the correct subkey is at position $i_0, i_1, \ldots i_{15}$ respectively. Thus, the more probable full key candidates are output first. Note that many key candidates will have equal probabilities Z.

A *side-channel adversary* outputs a list of key chunk candidates, sorted according to their probability of being the correct subkey k. More likely candidates appear first, and each candidate appears only once in the list.

In the Unified Framework introduced by Standaert et al. [107], two metrics are proposed to assess the strength of a side-channel adversary:

1. the **o-th order success rate** SR_0 is the probability that the correct subkey appears at position less or equal to o. If k is always ranked first, then for any $o \geq 1$, $SR_o = 1$.

2. the **guessing entropy** E is the expected rank of the correct subkey. Therefore, it is the average number of subkeys to be tried out before the correct subkey is identified. Again if k is always output first, then $E = 1$.

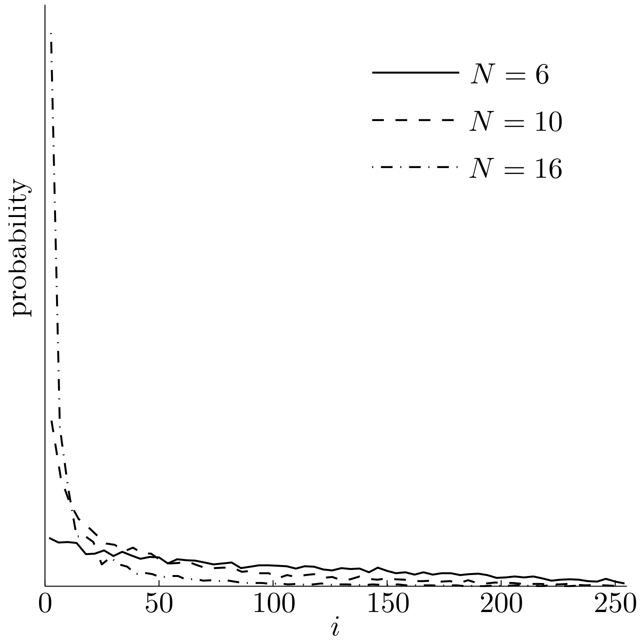


Figure 5.1: Empirical PMF for the rank i of the correct key byte candidate in the DPA sorted list. The DPA is performed using N traces.

These metrics also apply to full key PMFs. Therefore we can use them to compare our enumeration methods.

5.4 Lexicographical key enumeration

The straightforward strategy an adversary can adopt to combine subkey candidates is the *lexicographical* ordering, i.e. the order in which words are listed in a dictionary.

Assume $\mathcal{V}$ is a set of n-dimensional vectors: we have $\boldsymbol{v} = (v_1, v_2, \ldots, v_n) \in \mathcal{V}$ and each component v_i is defined over an ordered set. To keep the analogy with the dictionary and without loss of generality, we state that the larger the index i, the less significant the position in this ordering. We formally define the lexicographical order $\overset{lex}{\succ}$ with the following statement for two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{V}$:

$$\boldsymbol{u} \overset{lex}{\succ} \boldsymbol{v} \iff \exists i \in [\![1, n]\!],\ \forall j < i,\ (u_i < v_i) \wedge (u_j = v_j)$$

In the context of key enumeration, this ordering imposes us to restrain the search space from the start, otherwise the list of full candidates would be of the size of the search space which would not be sensible with lexicographical ordering (as we see later with extreme cases). Thus, we choose to consider only the top m candidates from every chunk distribution and a chunk candidate has a rank comprised between 1 and m. Since we deal with ranking vectors, $\mathcal{V}$ is the Cartesian product $[\![1, m]\!]^n$

where $[\![1, m]\!]$ is a subset of $\mathbb{N}$, thus it is an ordered set. The order relation $\overset{lex}{\succ}$ between two ranking vectors means: "is more likely than".

For example, we have

$$(1, 1, \ldots, 1, m) \overset{lex}{\succ} (1, 1, \ldots, 1, 2, 1)$$

and

$$(1, 1, \ldots, 1, m-1, m, m) \overset{lex}{\succ} (1, 1, \ldots, 1, m, 1, 1)$$

If we look at an extreme case such that:

$$(1, m, m, \ldots, m) \overset{lex}{\succ} (2, 1, 1, \ldots, 1)$$

this ordering may seem inappropriate for sorting key candidates by likelihood, especially when m is a bit large. However, the space complexity, namely $O(m^n)$ key candidates, increases so fast that only very small values for m are acceptable (in Section 5.6, our computation are limited to $m = 3$). In fact, when devising a sorting method, it turns out that a more efficient sorting is quickly more complex than the lexicographical ordering. For this reason, we keep it as our reference method for comparison.

Note that this ordering does not take into account the values of the PMF, only the ranks are considered.

5.5 Key enumeration using pairwise multiplications

In this section we present our method for key enumeration in a DPA attack. For simplicity in the description, we make the assumption that the position of the correct subkey will follow the same distribution for all subkeys (the key values change but the probability values remain the same).

In order to obtain a list of pairs of subkeys sorted according to the PMF of a single chunk of the key, we perform the following iterative steps:

1. Multiply the values of the subkey PMF pairwise (Figure 5.2). The obtained list is quadratically larger.

2. Sort this list.

3. If the list has become larger than a certain number T (dictated by the computing memory available), reduce the list to T elements.

The key values are tracked and combined along with the probability values. By iterating these steps, a list of full key candidates, sorted according to their likelihood, is eventually obtained. The steps of our key enumeration are described in Algorithm 7.

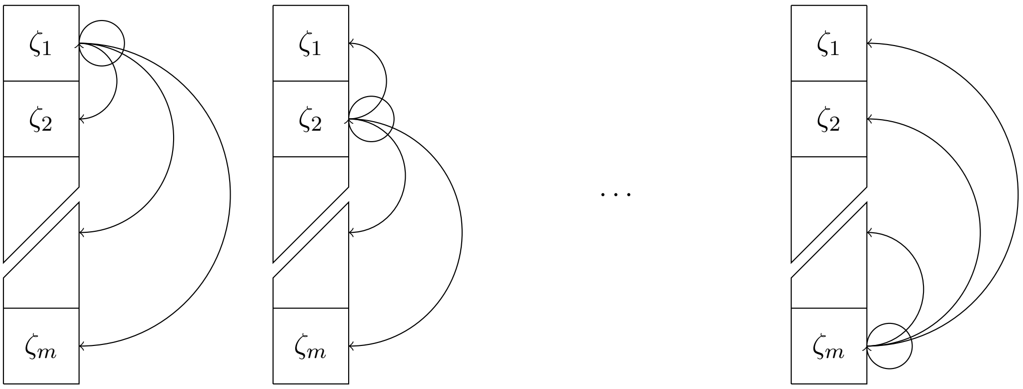


Figure 5.2: Pairwise computation of $\mathcal{Z} \times \mathcal{Z}$

Algorithm 7 Sorting of AES-128 key candidates using pairwise multiplications

```
Input: Subkey PMF $\mathcal{Z}$
for $r$ from 1 to 5 do
    for $i$ from 1 to $|\mathcal{Z}|$ do
        for $j$ from 1 to $|\mathcal{Z}|$ do
            $\eta_{i,j} = \zeta_i \times \zeta_j$                    ▷ Multiplicate
        end for
    end for
    $\mathcal{Z} \leftarrow \mathrm{Sort}(\{\eta_{i,j}\})$                    ▷ Sort
    if $r < 5$ then
        $\mathcal{Z} \leftarrow (\zeta_1, \ldots, \zeta_T)$                ▷ Truncate
    end if
end for
Output: Full key PMF $\mathcal{Z}$
```

t	`1-byte`	`lex`	`prw`
7	1.1828	1662500	17497
14	1.0124	266890	4.4721
21	1.0011	23677	1.1584
28	1.0002	4305.7	1.0275

Table 5.1: Expected rank of the correct key in single byte PMF (`1-byte`), in a 16-byte PMF sorted in lexicographical order (`lex`) and with pairwise multiplications (`prw`).

5.6 Experimental results

We implemented and applied our method in template-based DPA attacks against AES-128, in comparison with the lexicographical ordering.

The computation of the expected rank of the correct key in the PMF is indicated in Table 5.1. The number $t = 7, 14, 21$ or 28 corresponds to the number of power traces involved in the building phase of each template. For each of the lists of full key candidates, the success rate is always 1. Therefore, only the guessing entropy is relevant for comparison. The sizes of the considered lists are:

- in a single byte PMF: $2^8 = 256$
- in the lexicographical ordering: $3^{16} = 43\ 046\ 721$
- in the pairwise multiplication algorithm: $2^{26} = 67\ 108\ 864$.

We observe as expected that the guessing entropy when the `prw` algorithm is applied and $t = 14$ is much less than the guessing entropy when the `lex` ordering is applied and $t = 28$. Therefore, the application of an efficient sorting algorithm allows a reduction of the complexity of the building phase in a template-based DPA attack in comparison with a simple ordering.

5.7 Optimal key enumeration

A more efficient key enumeration method than our pairwise multiplication method was published by Standaert et al. [116]. We outline it in this section.

The method starts with the merging of two PMFs $\mathscr{Z}_1$ and $\mathscr{Z}_2$ which are in correspondence with two sorted lists of key bytes $(k_1^{(1)}, k_1^{(2)}, \ldots, k_1^{(256)})$ and $(k_2^{(1)}, k_2^{(2)}, \ldots, k_2^{(256)})$.

1. The most likely pair of bytes is naturally the pair of first elements from each list $k_1^{(1)}$ and $k_2^{(1)}$. It is the first pair of subkeys of the result list.
2. From it, a frontier set of pairs is built, composed of $(k_1^{(2)}, k_2^{(1)})$ and $(k_1^{(1)}, k_2^{(2)})$. The second most likely pair belongs to this set because the probabilities of the pairs in the frontier set will anyhow be higher than that of any subsequent

pairs. The two respective probabilities of the pairs in the frontier set are computed to determine which one must be the second element of the resulting list.

3. A new frontier set is built adjoining the result list, but composed of a single pair per line and column. In each row and column, the available element with the highest rank is always preferred. After multiplication of the probabilities among the pairs in the frontier set, the most likely pair is chosen as the third element of the result list.

4. In the further steps, Step 3 is iterated until the frontier set is empty, i.e. all possible pairs have been added to the result list.

In order to obtain a sorted list of full key candidates, the process is recursively applied. In the case of AES-128, it would be as follows:

1. The lists of 2-byte subkeys are obtained by merging two lists of single byte subkeys.

2. The lists of 4-byte subkeys are obtained by merging two lists of 2-byte subkeys.

3. The lists of 8-byte subkeys are obtained by merging two lists of 4-byte subkeys.

4. The lists of full keys are obtained by merging two lists of 8-byte subkeys.

For other key lengths, the process can be further applied or adapted.

We remark that, as for our method using pairwise computation, the enumeration is optimal (in the sense that no subkeys are omitted) as long as the intermediate lists are not truncated, which is not avoidable in practice. Although it is not possible to store a list with 2^{128} elements, the lists can be large enough for the probability of omitting the correct key to be negligible.

5.8 Conclusion

In this chapter we addressed the key enumeration problem in a divide and conquer side-channel attack: how to form full key candidates from the knowledge of the PMFs of the n key chunks. We put forward a key enumeration based on pairwise multiplication and recursive application. In comparison with a simple algorithm such as the lexicographical ordering, our method significantly improves the complexity of the “conquer” part of a template-based DPA attack. Otherwise presented, our method yields a comparable success in the overall template attack with a less complex template-building phase.

Bibliography

[1] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2006. Extended version available at `http://eprint.iacr.org/2006/138.pdf`. 13, 16, 26, 37, 38

[2] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Micro-architectural cryptanalysis. *IEEE Security & Privacy*, 5(4):62–64, 2007. 6

[3] Onur Acıiçmez and Çetin Kaya Koç. Microarchitectural attacks and countermeasures. *Cryptographic Engineering*, pages 475–504, 2009. 6

[4] Dakshi Agrawal, Selçuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan detection using IC fingerprinting. In *IEEE Symposium on Security and Privacy*, pages 296–310. IEEE Computer Society, 2007. 64, 65

[5] Ibrahim A. Al-Kadit. Origins of cryptology: The Arab contributions. *Cryptologia*, 16(2):97–126, 1992. 2

[6] ARM Ltd. Processors. `http://www.arm.com/products/processors/`, 2010. 15

[7] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerers apprentice guide to fault attacks. *IACR Cryptology ePrint Archive*, 2004:100, 2004. Available at `http://eprint.iacr.org/2004/100.pdf`. 66

[8] Lejla Batina, Benedikt Gierlichs, and Kerstin Lemke-Rust. Comparative evaluation of rank correlation based DPA on an AES prototype chip. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *ISC*, volume 5222 of *Lecture Notes in Computer Science*, pages 341–354. Springer, 2008. 49

[9] Georg T. Becker, Markus Kasper, Amir Moradi, and Christof Paar. Side-channel based watermarks for integrated circuits. In Jim Plusquellic and Ken Mai, editors, *HOST*, pages 30–35. IEEE Computer Society, 2010. 76

[10] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *EUROCRYPT*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 1994. 68

[11] Olivier Benoît and Thomas Peyrin. Side-channel analysis of six SHA-3 candidates. In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 140–157. Springer, 2010. Extended version available at `http://eprint.iacr.org/2010/447`. 44

[12] Daniel J. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2004. 15, 74

[13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak specifications. `http://keccak.noekeon.org/Keccak-specifications.pdf`, October 2008. 5

[14] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *ITCC'05*, volume 1, pages 586–591. IEEE, 2005. 13

[15] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 221–240. Springer, 2008. 67, 71, 72, 74, 78

[16] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997. 66, 74

[17] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. In Wiener [121], pages 216–233. 71

[18] Martin Boesgaard, Mette Vesterager, Thomas Pedersen, Jesper Christiansen, and Ove Scavenius. Rabbit: A new high-performance stream cipher. In Thomas Johansson, editor, *FSE*, volume 2887 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 2003. 71

[19] Andrey Bogdanov. Improved side-channel collision attacks on AES. In Carlisle Adams, Ali Miri, and Michael Wiener, editors, *SAC'07*, volume 4876 of *LNCS*, pages 84–95. Springer, 2007. 7, 16

[20] Andrey Bogdanov and Ilya Kizhvatov. Beyond the limits of DPA: Combined side-channel collision attacks. *IEEE Trans. Computers*, 61(8):1153–1164, 2012. 7

[21] Andrey Bogdanov, Ilya Kizhvatov, and Andrey Pyshkin. Algebraic methods in side-channel collision attacks and practical collision detection. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *INDOCRYPT'08*, volume 5365 of *LNCS*, pages 251–265. Springer, 2008. 7, 16

[22] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997. 66, 67, 74

[23] Joseph Bonneau. Robust final-round cache-trace attacks against AES. Cryptology ePrint Archive, Report 2006/374, 2006. `http://eprint.iacr.org/2006/374`. 13, 18, 19, 25, 33

[24] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *CHES'06*, volume 4249 of *LNCS*, pages 201–215. Springer, 2006. 15

[25] Christina Boura, Sylvain Lévêque, and David Vigilant. Side-channel analysis of Grøstl and Skein. In *IEEE Symposium on Security and Privacy Workshops*, pages 16–26. IEEE Computer Society, 2012. 45, 48

[26] Luca Breveglieri, Israel Koren, David Naccache, Elisabeth Oswald, and Jean-Pierre Seifert, editors. *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009.* IEEE Computer Society, 2009. 95

[27] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Joye and Quisquater [60], pages 16–29. 8, 12, 53

[28] Julien Brouchier, Nora Dabbous, Tom Kean, Carol Marsh, and David Naccache. Thermocommunication. *IACR Cryptology ePrint Archive*, 2009:2, 2009. Available at `http://eprint.iacr.org/2009/002`. 4

[29] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr, Luke O'Connor, Mohammad Peyravian, David Safford, et al. MARS: A candidate cipher for AES. In *In Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, 1998. 71

[30] Suresh Chari, Charanjit Jutla, Josyula R. Rao, and Pankaj Rohatgi. A cautionary note regarding evaluation of AES candidates on smart-cards. In *Second Advanced Encryption Standard Candidate Conference*, pages 133–147. Citeseer, 1999. 47

[31] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002. 7

[32] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Trans. Computers*, 53(6):760–768, 2004. Available at `http://eprint.iacr.org/2003/237`. 6

[33] Hamid Choukri and Michael Tunstall. Round reduction using faults. In Luca Breveglieri and Israel Koren, editors, *FDTC*, pages 13–24, 2006. 66

[34] Christophe Clavier and Kris Gaj, editors. *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*. Springer, 2009. 90, 93, 96

[35] Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In Clavier and Gaj [34], pages 156–170. 36

[36] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer, 2002. 17, 18

[37] Markus Dichtl. A new method of black box power analysis and a fast algorithm for optimal key search. *Journal of Cryptographic Engineering*, 1(4):255–264, October 2011. 80

[38] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. 4

[39] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on AES. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *ACNS*, volume 2846 of *Lecture Notes in Computer Science*, pages 293–306. Springer, 2003. 74

[40] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein hash function family. Available at: `http://www.skein-hash.info/sites/default/files/skein1.3.pdf`, 2010. 5, 45, 51

[41] PUB FIPS. 46-3: Data Encryption Standard, 1999. Available for download at `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`. 4, 17, 80

[42] PUB FIPS. 197: Specification for the Advanced Encryption Standard, 2001. Available for download at `http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf`. 4, 17, 18

[43] PUB FIPS. 180-4: Secure Hash Standard, 2012. Available at `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`. 5

[44] Defense Science Board Task Force. High performance microchip supply. Technical report, Office of the Secretary of Defense, 2005. Available for download at `http://www.acq.osd.mil/dsb/reports/ADA435563.pdf`. 64

[45] Jacques Fournier and Michael Tunstall. Cache based power analysis attacks on AES. In Lynn Margaret Batten and Reihaneh Safavi-Naini, editors, *ACISP'06*,

volume 4058 of *LNCS*, pages 17–28. Springer, 2006. xvii, 13, 18, 19, 20, 21, 22, 23, 25, 32, 42

[46] Jean-François Gallais, Johann Großschädl, Neil Hanley, Markus Kasper, Marcel Medwed, Francesco Regazzoni, Jörn-Marc Schmidt, Stefan Tillich, and Marcin Wójcik. Hardware Trojans for inducing or amplifying side-channel leakage of cryptographic software. In Liqun Chen and Moti Yung, editors, *INTRUST*, volume 6802 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2010. xvi, 10, 63, 70

[47] Jean-François Gallais and Ilya Kizhvatov. Error-tolerance in trace-driven cache collision attacks. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 222–232, Darmstadt, 2011. CASED. 9, 11

[48] Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall. Improved trace-driven cache-collision attacks against embedded AES implementations. In Yongwha Chung and Moti Yung, editors, *WISA*, volume 6513 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2010. Extended version available at `http://eprint.iacr.org/2010/408`. 9, 11

[49] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001. 13

[50] Henri Gilbert, Marc Girault, Philippe Hoogvorst, Fabrice Noilhan, Thomas Pornin, Guillaume Poupard, Jacques Stern, and Serge Vaudenay. Decorrelated Fast Cipher: An AES candidate. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, 1998. 71, 74

[51] Henri Gilbert and Helena Handschuh, editors. *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*. Springer, 2005. 94

[52] Brian Gladman. A specification for Rijndael. `http://gladman.plushost.co.uk/oldsite/cryptography_technology/rijndael/aes.spec.v316.pdf`, 2007. 71

[53] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In Donghoon Lee and Seokhie Hong, editors, *ICISC*, volume 5984 of *Lecture Notes in Computer Science*, pages 176–192. Springer, 2009. 6, 69, 76, 78

[54] Trusted Computing Group. TCG specification architecture overview (revision 1.4), 2004. Available for download at `http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf`. 64

[55] Tzipora Halevi and Nitesh Saxena. Eavesdropping over random passwords via keyboard acoustic emanations. *IACR Cryptology ePrint Archive*, 2010:605, 2010. Available at `http://eprint.iacr.org/2010/605`. 4

[56] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES smart card implementation resistant to power analysis attacks. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *ACNS*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252, 2006. 35

[57] Yier Jin, Nathan Kupp, and Yiorgos Makris. Experiences in hardware Trojan design and implementation. In Mohammad Tehranipoor and Jim Plusquellic, editors, *HOST*, pages 50–57. IEEE Computer Society, 2009. 64

[58] Yier Jin and Yiorgos Makris. Hardware Trojan detection using path delay fingerprint. In Mohammad Tehranipoor and Jim Plusquellic, editors, *HOST*, pages 51–57. IEEE Computer Society, 2008. 65

[59] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA cryptography specifications version 2.1, 2003. 76

[60] Marc Joye and Jean-Jacques Quisquater, editors. *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*. Springer, 2004. 89, 93

[61] Pascal Junod and Serge Vaudenay. Optimal key ranking procedures in a statistical cryptanalysis. In Thomas Johansson, editor, *Fast Software Encryption*, volume 2887 of *Lecture Notes in Computer Science*, pages 235–246. Springer Berlin / Heidelberg, 2003. 80

[62] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8:141–158, 2000. 15

[63] Auguste Kerckhoffs. La cryptologie militaire. *Journal des Sciences Militaires*, vol. IX:5–38, January 1883. 2

[64] Ilya Kizhvatov. Side channel analysis of AVR XMEGA crypto engine. In Dimitrios N. Serpanos and Wayne Wolf, editors, *WESS*. ACM, 2009. 49

[65] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. 3, 12

[66] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Wiener [121], pages 388–397. 6, 8, 12

[67] Valentin F. Kolchin, Boris A. Sevastyanov, and Vladimir P. Chistyakov. *Random Allocations*. V. H. Winston & Sons, Washington, D.C., 1978. 37

[68] NTT Secure Platform Laboratories. FEAL-NX specifications. `http://info.isl.ntt.co.jp/crypt/eng/archive/dl/feal/call-3e.pdf`. 51

[69] Xuejia Lai and James L. Massey. Markov ciphers and differential cryptanalysis. In Donald W. Davies, editor, *EUROCRYPT*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38. Springer, 1991. 71, 74

[70] Kerstin Lemke, Kai Schramm, and Christof Paar. DPA on n-bit sized boolean and arithmetic operations and its application to IDEA, RC6, and the HMAC-construction. In Joye and Quisquater [60], pages 205–219. 45, 47

[71] A.K. Lenstra. Memo on RSA signature generation in the presence of faults. *Private communication (available from the author)*, September 28, 1996. 68

[72] ARM Limited. ARM7TDMI technical reference manual (revision r4p1), November 2004. ARM Doc No. DDI 0210, Issue C, available for download at `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf`. 69

[73] Lang Lin, Wayne Burleson, and Christof Paar. MOLES: Malicious off-chip leakage enabled by side-channels. In *ICCAD*, pages 117–122. IEEE, 2009. 64

[74] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware Trojans through side-channel engineering. In Clavier and Gaj [34], pages 382–395. 64

[75] A.W. Machado. The Nimbus cipher: A proposal for NESSIE, September 2000. 71

[76] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards.* Springer Verlag, 2007. 7, 16, 18, 76

[77] Mitsuru Matsui. The first experimental cryptanalysis of the data encryption standard. In Yvo Desmedt, editor, *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1994. 80

[78] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *SAC'06*, volume 4356 of *LNCS*, pages 147–162. Springer, 2007. 15

[79] NXP B.V. LPC2114/2124 single-chip 16/32-bit microcontrollers. `http://www.nxp.com/documents/data_sheet/LPC2114_2124.pdf`, 2007. 19

[80] Olimex. LPC-H2124 header board for LPC2124 ARM7TDMI-S microcontroller. `http://www.olimex.com/dev/lpc-h2124.html`. 19

[81] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA'06*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. 15

[82] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES S-box. In Gilbert and Handschuh [51], pages 413–423. 44

[83] Elisabeth Oswald and Bart Preneel. A theoretical evaluation of some NESSIE candidates regarding their susceptibility towards power analysis attacks. Available at `https://www.cosic.esat.kuleuven.be/nessie/nessie/reports/phase2/kulwp5-022-1.pdf`, 2002. 47

[84] Dan Page. *A Practical Introduction to Computer Architecture*. Springer, 2009. 53

[85] Daniel Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical report CSTR-02-003, University of Bristol, 2002. 13, 15

[86] Jing Pan, Jasper G. J. van Woudenberg, Jerry den Hartog, and Marc F. Witteman. Improving DPA by peak distribution analysis. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 241–261. Springer, 2010. 81

[87] Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance (corresp.). *Information Theory, IEEE Transactions on*, 24(1):106–110, January 1978. 68

[88] Emmanuel Prouff. DPA attacks and S-boxes. In Gilbert and Handschuh [51], pages 424–441. 7, 44

[89] Emmanuel Prouff and Patrick Schaumont, editors. *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*. Springer, 2012. 95

[90] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *E-smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001. 13

[91] Chester Rebeiro and Debdeep Mukhopadhyay. Cryptanalysis of CLEFIA using differential methods with cache trace patterns. In *CT-RSA'11*, LNCS. Springer, 2011. 19

[92] Mathieu Renauld and François-Xavier Standaert. Algebraic side-channel attacks. In *Inscrypt*, pages 393–410, 2009. 7, 16

[93] Mathieu Renauld, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Algebraic side-channel attacks on the AES: Why time also matters in DPA. In *CHES'09*, volume 5747 of *LNCS*, pages 97–111. Springer, 2009. 7, 16

[94] R.L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6 block cipher. In *in First Advanced Encryption Standard (AES) Conference*. Citeseer, 1998. 71, 74

[95] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. 4, 74, 77

[96] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. Simple photonic emission analysis of AES - Photonic side-channel analysis for the rest of us. In Prouff and Schaumont [89], pages 41–57. 4

[97] Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos. Optical fault attacks on AES: A threat in violet. In Breveglieri et al. [26], pages 13–22. 66

[98] Jörn-Marc Schmidt and Marcel Medwed. A fault attack on ECDSA. In Breveglieri et al. [26], pages 93–99. 66

[99] Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar. A collision-attack on AES: Combining side channel- and differential-attack. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES'04*, volume 3156 of *LNCS*, pages 163–175. Springer, 2004. 7, 16

[100] Kai Schramm, Thomas Wollinger, and Christof Paar. A new class of collision attacks and its application to DES. In Thomas Johansson, editor, *FSE'03*, volume 2887 of *LNCS*, pages 206–222. Springer, 2003. 7, 16

[101] Beale Screamer. Microsoft's Digital Rights Management Scheme – Technical Details. `http://cryptome.org/ms-drm.htm`, October 2001. 71

[102] Adi Shamir and Eran Tromer. Acoustic cryptanalysis: On nosy people and noisy machines, 2004. Available at `http://tau.ac.il/~tromer/acoustic/`. 4

[103] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, vol. 28(4):656–715, 1949. 3

[104] Simon Singh. *The Code Book (Histoire des codes secrets)*. J.-C. Lattès (pour la traduction française), Paris, 1999. 2

[105] Sergei Skorobogatov and Christopher Woods. Breakthrough silicon scanning discovers backdoor in military chip. In Prouff and Schaumont [89], pages 23–40. 65

[106] Sergei Skorobogatov and Christopher Woods. In the blink of an eye: There goes your AES key. *IACR Cryptology Eprint Archive*, 2012:296, 2012. Available at `http://eprint.iacr.org/2012/296`. 65

[107] François-Xavier Standaert, Tal Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 443–461. Springer, 2009. 81

[108] Jacques Stern. *La Science du Secret.* Éditions Odile Jacob, Paris, 1998. 2

[109] Jon Stokes. *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture.* No Starch Press, San Francisco, CA, USA, 2006. 15

[110] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware Trojan taxonomy and detection. *IEEE Design & Test of Computers*, 27(1):10–25, 2010. 64, 69

[111] Kris Tiri and Ingrid Verbauwhede. Securing encryption algorithms against DPA at the logic level: Next generation smart card technology. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 2003. 44

[112] Randy Torrance and Dick James. The state-of-the-art in IC reverse engineering. In Clavier and Gaj [34], pages 363–381. 65

[113] Michael Tunstall, Neil Hanley, Robert P. McEvoy, Claire Whelan, Colin C. Murphy, and William P. Marnane. Correlation power analysis of large word sizes. In *IET Irish Signals and System Conference*, pages 145–150, 2007. 53

[114] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the Advanced Encryption Standard using a single fault. In Claudio Agostino Ardagna and Jianying Zhou, editors, *WISTP*, volume 6633 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2011. 74

[115] Praveen Kumar Vadnala, Jean-François Gallais, and Arnab Roy. Full key recovery attacks on modular addition: An application to Threefish. In *7-th Workshop on Embedded Systems Security 2012 (WESS 2012)*, Tampere, Finland, October 2012. 10, 43

[116] Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. *IACR Cryptology ePrint Archive*, 2011:610, 2011. Available at `http://eprint.iacr.org/2011/610`. 81, 85

[117] Adam Waksman and Simha Sethumadhavan. Tamper evident microprocessors. In *IEEE Symposium on Security and Privacy*, pages 173–188. IEEE Computer Society, 2010. 65

[118] Jörg Walter. Fhreefish – fast AVR 8-bit implementation of Threefish and Skein. `http://www.syntax-k.de/projekte/fhreefish/`. 58

[119] David J. Wheeler and Roger M. Needham. TEA, a tiny encryption algorithm. In Bart Preneel, editor, *FSE*, volume 1008 of *Lecture Notes in Computer Science*, pages 363–366. Springer, 1994. 51, 54

[120] David J. Wheeler and Roger M. Needham. TEA extensions. Technical report, Computer Laboratory, University of Cambridge, October 1997. 54

[121] Michael J. Wiener, editor. *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*. Springer, 1999. 88, 92

[122] Francis G. Wolff, Christos A. Papachristou, Swarup Bhunia, and Rajat Subhra Chakraborty. Towards Trojan-free trusted ICs: Problem analysis and detection scheme. In *DATE*, pages 1362–1365. IEEE, 2008. 64, 65, 69

[123] Xin-Jie Zhao and Tao Wang. Improved cache trace attack on AES and CLEFIA by considering cache miss and S-box misalignment. Cryptology ePrint Archive, Report 2010/056, 2010. Available at `http://eprint.iacr.org/2010/056`. 18

[124] Xinjie Zhao, Fan Zhang, Shize Guo, Tao Wang, Zhijie Shi, Huiying Liu, and Keke Ji. MDASCA: An enhanced algebraic side-channel attack for error tolerance and new leakage model exploitation. In Werner Schindler and SorinA. Huss, editors, *Constructive Side-Channel Analysis and Secure Design*, volume 7275 of *Lecture Notes in Computer Science*, pages 231–248. Springer Berlin Heidelberg, 2012. 7

[125] Michael Zohner, Michael Kasper, and Marc Stöttinger. Butterfly-attack on Skein's modular addition. In Werner Schindler and Sorin A. Huss, editors, *COSADE*, volume 7275 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2012. 45, 48

[126] Michael Zohner, Michael Kasper, Marc Stöttinger, and Sorin A. Huss. Side channel analysis of the SHA-3 finalists. In Wolfgang Rosenstiel and Lothar Thiele, editors, *DATE*, pages 1012–1017. IEEE, 2012. 45

List of publications

Jean-François Gallais, Arnab Roy and Praveen Kumar Vadnala. Full key recovery attacks on modular addition: an application to Threefish. Proceedings of *WESS 2012*.

Jean-François Gallais and Ilya Kizhvatov. Error-tolerance in trace-driven cache collision attacks. Proceedings of *COSADE 2011*.

Jean-François Gallais, Johann Großschädl, Neil Hanley, Markus Kasper, Marcel Medwed, Francesco Regazzoni, Jörn-Marc Schmidt, Stefan Tillich, Marcin Wójcik. Hardware Trojans for inducing or amplifying side-channel leakage. Proceedings of *INTRUST 2010*, volume 6802 of *LNCS*, pages 253–270. Springer, 2010.

Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall. Improved trace-driven cache-collision attacks against embedded AES implementations. Proceedings of *WISA 2010*, volume 6513 of *LNCS*, pages 243–257. Springer, 2011. Extended version available at `http://eprint.iacr.org/2010/408`.