Université catholique de Louvain
Faculté des Sciences Appliquées
Laboratoire de Microélectronique
UCL Crypto Group

Cache-based Vulnerabilities and SPAM analysis

Michael Neve

Thèse soutenue en vue de l'obtention du grade de Docteur en Sciences Appliquées

Louvain-la-Neuve, Belgique
Juin 2006

Abstract

Two very different problems are tackled in this dissertation. In the first part, we analyze new side channels that exploit the memory architecture of current processors; the cache in particular. By nature, the cache mechanism presents different timings in the memory accesses: in the best case, the requested data is held by the cache and the access time is short (cache-hit), otherwise the data must be fetched from a higher memory level and the access time is longer (cache-miss). In cache-based attacks, a program with some secret data is executed and an attacker exploits the timing differences in order to discover this secret data. New cache-based side channel attacks are presented for AES and software countermeasures are given.

The second part presents our analysis of spam mail. The purpose is to quantify the effectiveness of email address protections against the gathering tools and techniques of spammers. With the help of interconnected honey pots, we analyze aspects of the spammers' behavior.

Résumé

Deux thématiques très différentes sont abordées dans cette these. Dans une première partie, nous étudions une nouvelle classe d'attaques par canal caché qui tirent avantage de l'architecture de la mémoire des processeurs actuels et plus particulièrement la cache. Par nature, le mécanisme qui régit la cache entraine des différences dans les temps d'accès à la mémoire : d'un côté, si la donnée se trouve dans la cache, le temps d'accès est court; mais dans le cas contraire, la donnée est ramenée d'un niveau de mémoire plus élevé et cette opération nécessite plus de temps. Dans les attaques par cache, un programme manipulant une donnée secrete est executé sur un processeur et l'attaquant utilise ces différences de temps pour déduire cette donne secrète. De nouvelles attaques par cache contre l'AES sont présentées et des contre-mesures logicielles sont proposées.

La seconde partie détaille notre analyse des pourriels. L'objet de notre travail est de quantifier les performances des protections d'adresses électroniques, contre les techniques de récuperation d'adresses. De plus, nous étudions plusieurs aspects de ces mêmes techniques.

Acknowledgments:

I am grateful to Jean-Jacques Quisquater for convincing me that the UCL Crypto Group was the right place to set out on a PhD, and for helping me fully benefit of those years. I am thankful to Jean-Pierre Seifert and Jean-François Dhem who accepted to join the thesis committee and gave numerous useful comments on this work. I particularly thank Jean-Pierre for inviting me to an internship at Intel, for many fruitful discussions and judicious advices and for showing me it is a great place to work. Thanks to Jean-Pierre, Raj Yavatkar and Flemming Andersen for having turned the internship and the full-time offer into reality. Thanks to Çetin K. Koç, Jean-Didier Legat and Alphone Magnus for accepting to be part of the jury thesis.

I would also like to mention the F.N.R.S. for its financial support throughout a F.R.I.A. grant; the European Commission and Intel Corp. for temporary financial support.

I am also grateful to my co-authors, in alphabetical order: Lilian Bohy, Ernie Brickell, Mathieu Ciet, Damien Giry, Gary Graunke, Guerric Meurice de Dormale, Eric Peeters, Jean-Jacques Quisquater, David Samyde, Jean-Pierre Seifert, François-Xavier Standaert and Zhenghong Wang.

Thanks to Ernie, Gary and Jean-Pierre for sharing their expertise in the various aspects of computer security.

Thanks to Brigitte Dupont, Christophe François, François Hubin, Damien Giry, François Mace, Pascal Maes and Geoffroy Simon for the strong teamwork into the IT system providing a efficient, reliable and secure network for the Microelectronics Lab. They do not receive often enough the praise that they deserve.

I am thankful to Sylvie Baudine who checked the *Americannibalism* of most of my works.

Many thanks to Damien Giry for numberless hours of works, ideas, discussions. Thanks for his unlimitedness.

I am particularly thankful to my family and Nathalie for their day after day unconditional support.

Contents

Context and Motivation 1

Part 1. Cache Attacks: vulnerabilities and mitigations 7

Chapter I. Two Cryptographic Algorithms 9
- I.1. Introduction 9
- I.2. AES 12
- I.3. RSA 16
- I.4. Summary 19

Chapter II. Cache Architecture 21
- II.1. Microprocessor 21
- II.2. Hiding the main memory's latency 23
- II.3. Principle of locality 24
- II.4. Model of memory hierarchy 24
- II.5. Fully associative cache 24
- II.6. Direct-mapped cache 25
- II.7. Set associative cache 26
- II.8. Replacement policy 27
- II.9. Specialized caches 28
- II.10. Examples 30
- II.11. Summary 34

Chapter III. From confinement to side channels 37
- III.1. Time-driven methodology. 40
- III.2. Trace-driven methodology. 41

Chapter IV. Time-driven attacks 43
- IV.1. Preliminary note 43
- IV.2. Tsunoo *et al.* 43
- IV.3. Bernstein's Attack 44
- IV.4. Experiments 51
- IV.5. Round 2 Attack 74
- IV.6. Summary 77

Chapter V. Access-driven Attacks 81
- V.1. Page's theoretical attack 81

V.2. Percival's Attack on RSA 82
V.3. Attacks of Osvik *et al.* 84
V.4. Attacks on single-threaded processors 86
V.5. Multi-threaded *vs* single-threaded 86
V.6. Analysis of the last round 88
V.7. Average number of accesses for the last round 90
V.8. Resolution 90
V.9. Non-elimination and elimination methods 91
V.10. Practical considerations 97
V.11. Other attacks 98
V.12. Summary 99

Chapter VI. Mitigations of Cache-based Side Channels on AES 101
VI.1. Introduction 101
VI.2. Discussion of a side channel adversarial threat model 104
VI.3. Mitigations against cache-based side channel attacks 105
VI.4. Practical results of our mitigations 108
VI.5. Mitigation codes 109
VI.6. Conclusions and recommendations for further research 113

Part 2. SPAM analysis and prevention 115

Chapter VII. SPAM 117
VII.1. Introduction 117
VII.2. Phase 1: System Description 122
VII.3. Phase 1: Results 130
VII.4. Phase 2: Disturbing the System 134
VII.5. Phase 2: Results 136
VII.6. Phase 3: Impact of Spyware and Variation of e-Ad-dresses 140
VII.7. Phase 3: Results 141
VII.8. Conclusion 143

Conclusions 147

Chapter VIII. Conclusions and Perspectives 149

Bibliography 153

Appendix A. Publication list 161
A.1. Publications 161
A.2. Under submission 162

Context and Motivation

Introductory courses on cryptology usually start by presenting three popular protagonists: Alice, Bob and Eve. Let us perpetuate this tradition to display the framework of the present dissertation. Alice and Bob want to have a private conversation without Eve being able to understand any part of it. The problem is that all the accessible communication channels are public: let us first imagine that they are in the same room and that Eve has an extraordinarily acute hearing. Therefore she could eavesdrop any verbal conservation between Alice and Bob.

Here cryptography can help the two of them getting the privacy they are seeking.

Symmetric cryptography algorithms provide them ways to encrypt their communication with a *secret* key. Alice and Bob share the secret key and therefore they can encrypt a message to the attention of each other and decrypt the messages each of them receives. On the contrary, Eve does not have the secret key and thus she can neither encrypt a message of her own nor can she decrypt the messages she captures.

Consider the following example. Alice and Bob receive a box that can be securely opened and closed with a numerical combination (like a PIN number). Bob puts a message inside the box, locks the box with the secret combination and hands the box to Alice. Alice uses the same combination to open the box and reads Bob's message. Let us introduce some cryptographic terminology: the message is called the *plaintext*, the locked message is the *ciphertext*, the secret combination is the *secret key* and the method to lock the box, *i.e.* inputting the combination through the numeric pad, is the *cipher*.

Symmetric algorithms would be sufficient for an important number of applications without the key agreement problem. Prior to encrypting their communication (locking the box), Alice and Bob have to agree on the secret key. They cannot discuss nor transmit the key in clear without Eve eavesdropping it. They have to come with a system that enables them to securely transit or agree on the secret key.

Asymmetric cryptography offers solutions to this problem. The principle behind those solutions is based on a concept different than for the symmetric ciphers. Here a pair of keys is generated for Alice: a *public* key and a *private* key. Both keys are linked to each other by mathematical properties. Let us illustrate this new concept by another box. This one has a padlock that can be locked by only one specific key and only one other specific key is able to unlock it. The box contains the message; when the padlock is locked the message is protected until the padlock gets opened with the private key. The method to lock and unlock the padlock is the *cipher* and would be, in this example, to turn one key in the padlock's keyhole.

The public key is accessible to anybody while the private key is only known by Alice. Bob and Eve can use Alice's public key to encrypt messages to Alice's intention. However no decryption is possible without knowing Alice's private key; Alice is then the only person able to decrypt the messages encrypted by her public key.

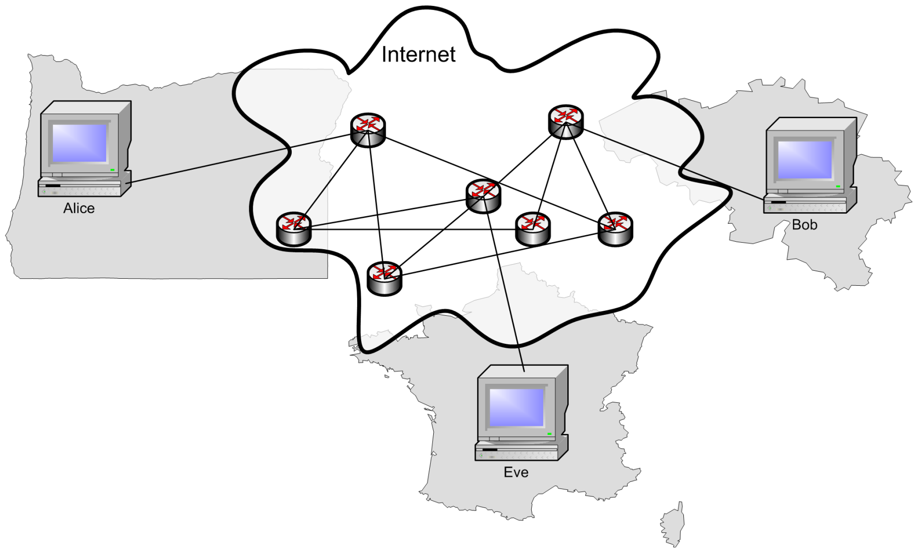
Thus, Bob selects a secret key to be used for their private communication and encrypts it with Alice's public key. The ciphertext can be transmitted over the insecure channel to Alice. Eve can eavesdrop but only Alice could decipher the ciphertext to obtain the secret key. There are then three keys used in this scenario: a secret key selected by Bob for the symmetric scheme and Alice's private and public keys used to securely transmit the secret key from Bob to Alice[1].

So far, Alice and Bob have an algorithm to securely communicate with each other: thanks to an asymmetric cryptographic algorithm, they can agree upon a secret key; they then use a symmetric cipher to crypt their communication with this secret key. Unfortunately for Eve, she could not eavesdrop any part of the key agreement process nor of Alice and Bob's communication. However the game is not over for Eve yet.

Alice and Bob are indeed facing practical considerations. Let us first change the environment a little to get closer to practical cases. We place Alice, Bob and Eve in three completely separated rooms. Each room has a computer wired to a network (*e.g.* the Internet). It is now their only mean of communication but it is not private: every message sent on the network is broadcast on (at least) all three computers. Each computer has also a compiler that they can use to write software.

Bob implements a software version of the symmetric and asymmetric algorithms discussed above and sends his program on the network. With

[1]This easy scheme can get really complicate in scenarios where Eve could pretend to be Bob and sends, on Bob's behalf, a secret key she has chosen. Alice cannot be sure of the identity of the sender. To solve this issue, Bob also needs to have a pair of public and private keys of his own, so that Alice and himself can authenticate each other. We do not detail any further such protocols as they are out of our scope of this introduction.



Bob's software, Alice creates a pair of public and private keys and broadcasts her public one on the network. With this latter, Bob encrypts a secret key of his choice and sends the ciphertext back on the network. Only Alice is able to decrypt it and get the secret key. Eventually, Alice and Bob encrypt their messages with the secret key that they share. In her room, Eve only sees the encrypted messages going back and forth and has no clue on the content of Alice and Bob's communication.

However, Eve does not give up. She analyzes the code of Bob's software, hoping to find flaws. She first looks at the key size: she could indeed program a brute-force program that tries all possible keys within that size; if the key is not long enough, her program could find the correct key in an acceptable time. But Bob did a great job: from `www.keylength.com` he selected appropriate key parameters. Even by using the processing power of all connected computers, Even would not discover the key in (thousands of) years.

More single-minded than ever, Eve then explores *cryptanalysis*: she studies the mathematical properties of the cryptographic algorithms to discover any theoretical or computational weakness. In vain ...

Eve is getting more and more curious about Alice and Bob's conversation. Her last hope is *side-channels*. These techniques take advantage of information aside of theoretical model of a system. For example, they analyze the execution time of the encryption, its power consumption, its electromagnetic radiations, etc. in other words, all physical information leaking from a particular implementation. Eve knows that it has been applied on smart card security for more than a decade now and it also showed promising results on general-purpose processors.

With the help of the present thesis, Eve finally gets a chance to sneak into Alice and Bob's chat.

∵

Even if naive, our scenario involving Alice, Bob and Eve summarizes a practical problem that impacts countless today's and tomorrow's applications: secure –but yet efficient– communications. Remote banking, data encryption, telecommunication and many more require at some point data to be processed through a computer. It is then crucial to make sure the computer is part of the security chain and that it processes the data without leaking information. The chain is only as secure as its least secure link. The purpose of the thesis is to investigate hardware security at the processor's level to guarantee that the computer is not the weakest link.

This thesis is split into three parts. Two problems of computer security are investigated into the first two parts. On one hand, we are facing a practical problematic of actual processors: elements of the architecture that bring flexibility and allow efficient utilization of the resources, are demonstrated to open security breaches from which secret information can be extracted. This issue required a delicate study to understand the problem and the role of the incriminated elements, to discover the potential of the attacks and find effective countermeasures.

On the other hand, we tackle a situation of computer science that also concerns many people and where economical aspects are at stake: although spam is often considered as the other side of the Internet coin, we believe that it can be defeated and avoided. A increasing number of researches for example explores the ways cryptographic techniques can prevent spams from being spread. We concentrated on studying the behavior of the spammers to understand how e-mail addresses can be prevented from being gathered.

Even if orthogonal, both parts tackle practical problems and their results can be directly applied. This is the reason why these two thematics appears in this dissertation, whereas more theoretical or less applicative works of mine are not covered here.

The conclusions of this work are given in the third part.

We give here a more precise layout of this dissertation by detailing the content of each chapters.

The first part focuses on cache-based side channels. Therefore Chapter I introduces the cryptographic algorithms utilized in this part, namely the Advanced Encryption Standard (AES) and the RSA. We define the notations used in the rest of the thesis and explain the principles of those algorithms. This introduction provides a sufficient background to understand the next chapters. References are however given for any further details.

Chapter II presents the cache of processors. We detail the central role it plays in nowadays processors and demonstrate the aggressive trade-off it supports, between speed and capacity. We not only show that without the cache, the processors would offer terrible performances but we also highlight that the price to pay is non-constant execution time, which can be especially harmful in security applications. Note that this work is essentially oriented toward processors of the PC framework, in contrast with processors for hand-held devices. When possible, we draw a parallel between this work and others applications. Extension of this work to the test of an exhaustive list of processors would be interesting but is out of the scope of this thesis, our purpose being to unveil the vulnerabilities and propose mitigations strategies.

Further, in Chapter III, we briefly retrace the origins of cache-based side channels. In particular, we show it started in the 1970s (and even before) with the confinement problem: the first published characterization in the framework of covert channels. In this context, two supposedly isolated entities use a service in a special manner in order to communicate (although the service is not intended for communication). The entities must collude and agree on the communication protocol. We then show that in particular cases like cryptographic applications, no collusion is necessary since the algorithms used are generally of public knowledge. Therefore, one can spy a crypto-process and deduce information about its secrets through the observation of leakages. And that is why the cache is under investigation in this thesis. It is then a case of side channel.

After those introduction chapters, we investigate in Chapter IV a first kind of cache-based side channel: time-driven attacks. Because of the intricate behavior of a processor and limited resources of the cache, it is extremely hard to write constant-time software. This is particularly true with cryptographic applications that often rely on large precomputed data and pseudo-random accesses. The principle of time-driven attacks is to analyze the overall execution time of a cryptographic process and extract timing profiles. We show that in the case of AES those profiles are dependent on the memory lookup, *i.e.* the addition of the plaintext and the secret key. Correlations between some profiles with known inputs and some with partially unknown ones (known plaintext but unknown secret key) lead to the recovery of the secret key.

Chapter V then details access-driven attacks: another kind of cache-based side channel. This case relies on stronger assumptions regarding the attacker's capacities: he must be able to run another process, concurrent to the security process. Even if the security policies prevent the so-called *spy* process from accessing directly the data of the *crypto* process, the cache is shared between them and its behavior can lead the spy process to deduce the secrets of the *crypto* process.

Mitigations are presented in Chapter VI. Several ways are explored and stated depending on the security level to reach and on the attacker's capabilities. The respective performances of the mitigations are given. This Chapter is however oriented toward software mitigations as they can be directly applied to patch programs and reduce the cache leakage.

The second part is composed of the single Chapter VII that presents a study we performed on the early stages of spam and on the characterization of the general behavior of the spammers. Our motivation is to define the adequate countermeasures to prevent an email address from being captured by spammers. One will not be receiving spam if the spammers do not know their address in the first place. We extend this topic to the analysis of the capacities of the spammers. We detail our honey pots and give our conclusions on the behavior, in order to propose *ad hoc* address protections. The motivation for this work was to produce and make available quantitative results to efficiently prevent spam, as well as to provide a better understanding of the behavior of spammers.

This second part contrasts with the first one and addresses an aspect of security on a different level. The spam problematic is also becoming an increasing economic issue for government, Internet service providers and end-users. Eventually solving the puzzle of spam will require the combination of efforts from different levels. Results like the ones we propose help the community to better understand the spammers' strategy, in order to better defend/immune ourself and fight against them.

Eventually, the last Chapter concludes this thesis and discusses possible perspectives.

Most of the results presented here have been published or have been submitted. However, it only covers a limited part of the topics studied during the Ph.D. We refer to Appendix A for the list of publications.

Part 1

Cache Attacks: vulnerabilities and mitigations

CHAPTER I

Two Cryptographic Algorithms

Abstract: The Chapter introduces the two cryptographic algorithms used in the rest of this dissertation: we present the Advanced Encryption Standard (AES), a symmetric algorithms that was chosen to replace the DES [77] as the encryption standard, and RSA, a asymmetric algorithms widely used for encryption, signature and authentication in electronic commerce and communications. We focus on the essential aspects and on some notations. Many references exist in the literature with the purpose of thoroughly describing the schemes; we refer the reader to them for more information.

I.1. Introduction

In 1883, the Dutch linguist Auguste Kerckhoffs von Nieuwenhof [48, 49] stated that a cryptosystem should remain secure even if everything about the system, except the key, enters common knowledge (this is known as Kerckhoffs' principle). Claude Shannon also formulated this idea: "[Assume] the enemy knows the system being used" (Shannon's maxim). So the key should be considered as the only secret part. By this way,

(1) one can compute an adequate key size in order to be way out of an attacker's computational reach,
(2) considering that key is discovered, one can simply change the key to be protected again.

There is a third advantage when applying Kerckhoffs' principle: standardized algorithms have been intensively and independently studied by the community and most security aspects have been examined. For example, the Advanced Encryption Standard (AES) is the result of a long selection among various candidate ciphers. This whole philosophy goes against the trend that many businesses or norms have overexploited: *security through obscurity*.

If the whole system is secret, as soon as the system (or a critical part of it) is discovered and the secrecy is in jeopardy, the system must be replaced on every compromised device or token. This becomes a major issue for medium to large scale systems. The Content Scrambling System (CSS) is one (of the too many) example(s) of high profile failures[1].

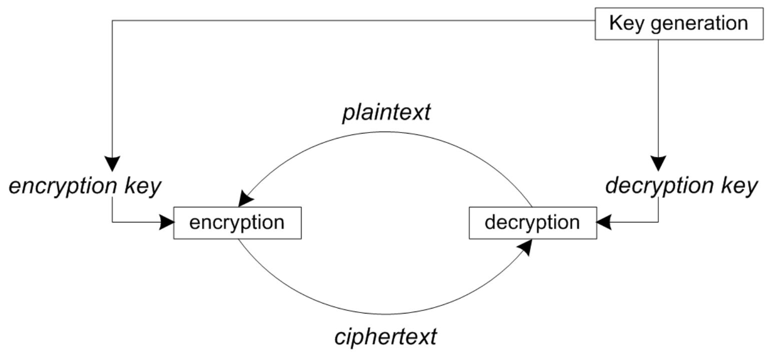
Hollywood uses CSS to encrypt DVD. The play back of the content was supposed to be impossible without the disk. However CSS was designed with a 40-bit key length to comply with US government export regulation and the system was (and still is) then vulnerable to brute-force attacks. Moreover, in 1999, CSS was broken with lower complexity (2^{25}) and a description algorithm was released.

Hollywood was hoping that this proprietary algorithm would not be discovered... in vain.

Instead of using proprietary pseudo-secure algorithms, we use commonly used algorithms throughout this work. This Chapter focuses on the description of two cryptographic algorithms: the AES and the RSA. In the present dissertation, we will consider implementation aspects of those two schemes and therefore we introduce here useful notations.

The first algorithm, AES, is a symmetric scheme whereas RSA is an asymmetric one. Let us briefly introduce these concepts. The purpose of encryption schemes is to make private communications possible over an insecure channel. In particular, a sender (Alice) transmits secret information to a receiver (Bob) over a communication mean that may be tapped by an adversary (Eve).

The message that Alice wants Bob to receive is called the *plaintext*. Alice transforms the plaintext thanks to the cryptographic scheme (*encryption*), that outputs the *ciphertext*: the ciphered message. The ciphertext must be unintelligible to anyone except the intended receiver. Therefore, there need to be something different between Bob and Eve. We know that a secret encryption scheme is not an option, under Kerckhoffs' principle. The solution is that only Bob knows the *key* that enables him to transform the ciphertext back to the plaintext (*decryption*).



[1]Other examples: GSM cell phones [11], Windows XP product activation, RIAA digital music watermarking, Adobe eBooks, ...

The cryptographic scheme is composed of those encryption and decryption algorithms, as well as a probabilistic *key generation* algorithm (as depicted in the sketch above). The latter produces a pair of keys, one to encrypt the plaintext into the ciphertext and another one to decrypt the ciphertext into the original plaintext. The keys must be somehow related in order to fulfill this property. Two relations between the keys are possible and we usually make the following distinction:

- Identical keys: the same key is used to encrypt and decrypt the communication. Clearly, the possession of the key should be limited to legitimate communicating persons. Therefore the key is called *secret key* and schemes using this relation are designated as *symmetric schemes* (Figure 1).

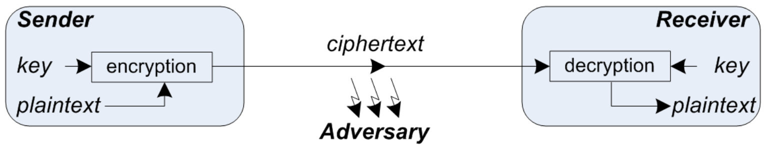


Figure 1: Symmetric scheme.

- Different keys: the encryption key is different than the decryption key. Here the decryption key alone needs to be secret and the receiver is the only party required to possess it. Therefore, a pair of keys is generated by the receiver. He keeps the decryption key for him-self (*private key*). And he may send the encryption key in clear to the sender (*public key*). Here any party can send an encrypted message to the receiver, using the public key. However, it is (nearly) impossible to deduce the private key from the public one. The adversary is then unable to get anything out of the ciphertext, even if he knows the public key. Figure 2 illustrates asymmetric crypto schemes.

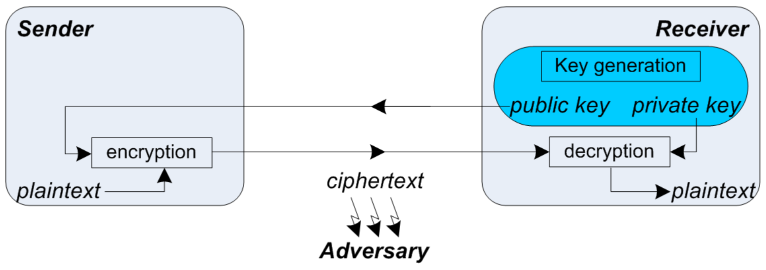


Figure 2: Asymmetric schemes.

The following Sections cover one type of symmetric scheme, AES, and one type of asymmetric scheme, RSA.

I.2. AES

I.2.1. Historical context

The *Advanced Encryption Standard* (AES) was introduced in 2001 by the National Institute for Standards and Technology (NIST) [78]. It replaced the Data Encryption Standard (DES [77]), that was in use since 1976, as standard recommendation of the US *Federal Information Processing Standards* (FIPS). The European project Nessie [91] surveyed the protocols and algorithms for signature, integrity and encryption and concluded in giving a broader recommendation panel than only AES as symmetric encryption algorithm. AES is the result of a selection between several candidates. The contest started in 1997 and the final round was in 2000. The winner was Rijndael, proposed by J. Daemen and V. Rijmen [25, 30]. AES is probably today's most commonly used block cipher in the Internet and software market[2]: applications include disk and file encryptions, wireless LAN security (IEEE 802.11i [43]), IPSec (standard for securing Internet protocol at the network layer [96]), Transport Layer Security (successor of SSL [95]), VoIP security, smart cards, microprocessors, and many others.

A strong point of AES is clearly its simplicity that enables efficient implementations over various platforms: from small 8-bit to modern 32-bit [65] or 64-bit microprocessors [63], or even on dedicated hardware (ASIC) [89, 102] or FPGA [23, 62, 66, 67, 89, 90, 114]. It is also applicable to different constraints: high throughput, low area, low power consumption, etc.

I.2.2. Description

We only recall specific points and give some notations needed throughout the present dissertation.

AES is a block cipher with 128-bit input blocks[3] represented by

$$\mathbf{p} = (p_0 \, , \ldots , \, p_{15})$$

and a key size of 128, 192 or 256 bits represented by

$$\mathbf{k} = (k_0 \, , \ldots , \, k_{n-1})$$

with $n = 16$, 24 or 32. In this document, we will use $n = 16$, but it is straightforward to extend the definitions to longer key sizes. For

[2]Banking and telecomminucation application

[3]However, Rijndael allows a variable block length and a variable key size. Both can be independently chosen as multiples of 32-bit with a minimum of 128-bit and a maximum of 256-bit. The AES standard normalizes the inputs' size to 128 bits.

encryption, the inputs are the plaintext and the key, and the output is the ciphertext. For decryption, the ciphertext and the key are the inputs and the result is the plaintext.

AES is a key-iterated block cipher (Figure 3): it is composed of a key schedule and repeated round transformations. The key schedule generates the succession of *round keys* that will be used in the successive rounds.

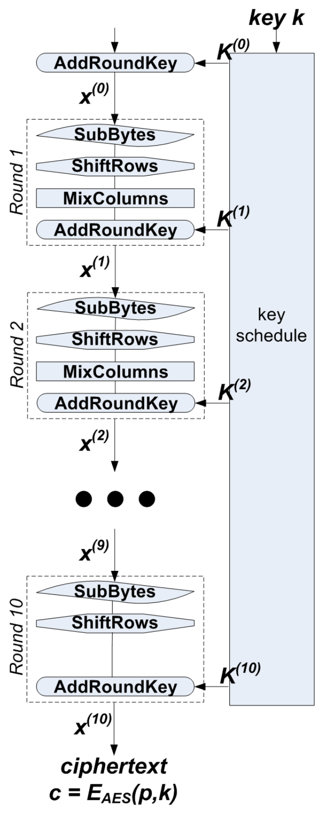


Figure 3: Representation of an AES encryption. The left part details the succession of rounds and the right one represents the key schedule. The plaintext **p**, the key **k** and the ciphertext **c** are 128-bit long.

A 16-byte key

$$\mathbf{k} = (k_0\,, \ldots,\, k_{15})$$

is expanded by `KeyExpansion` into 11 round keys

$$\mathbf{K}^{(r)} = (K_0^{(r)}, \ldots,\, K_{15}^{(r)})$$

for $r = 0, \ldots, 10$; with $\mathbf{k} = \mathbf{K}^{(0)}$ [4]. In addition, the `KeyExpansion` operation is reversible: in particular for 128-bit keys, if one knows the value of one round key $\mathbf{K}^{(i)}$, he can compute any other one.

An encryption with AES is operated as follows. After an initial key addition `AddRoundKey`, AES performs 10 successive rounds where `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` are applied to a state[5]. A state is defined as $\mathbf{x}^{(r)} = (x_0^{(r)}, \ldots, x_{15}^{(r)})$ and it is the result of the r-th `AddRoundKey`. The initial state is obtained by the first `AddRoundKey`

$$x_{j,i}^{(0)} = p_{j,i} \oplus k_j.$$

We then introduce the r-th round of a plaintext

$$\mathbf{p}_i^{(r)} = (p_{0,i}^{(r)}, \ldots, p_{15,i}^{(r)})$$

as input of the r-th `AddRoundKey`

$$x_{j,i}^{(r)} = p_{j,i}^{(r)} \oplus K_j^{(r)}.$$

An encryption of plaintext $\mathbf{p}$ by AES with key $\mathbf{k}$ produces a ciphertext $\mathbf{c}$, denoted as $\mathbf{c} = E_{AES}(\mathbf{p}, \mathbf{k})$.

We intentionally do not give any detail on the decryption part of AES, as it it not necessary for the dissertation; refer to [25] for more details.

I.2.3. OpenSSL

OpenSSL [79] is a popular software library which provides AES functionalities. Let us elaborate on this particular implementation. It performs the round operations with a granularity of a word (4 bytes). For each round r, its state word

$$\mathbf{x}_i^{(r)} = (x_{4i}^{(r)}, x_{4i+1}^{(r)}, x_{4i+2}^{(r)}, x_{4i+3}^{(r)})$$

with $i = 0, \ldots, 3$, is generated as:

$$\begin{aligned}
\mathbf{x}_0^{(r)} &= T_0\left[x_0^{(r-1)}\right] \oplus T_1\left[x_5^{(r-1)}\right] \oplus T_2\left[x_{10}^{(r-1)}\right] \oplus T_3\left[x_{15}^{(r-1)}\right] \oplus \mathbf{K}_0^{(r)}\\
\mathbf{x}_1^{(r)} &= T_0\left[x_4^{(r-1)}\right] \oplus T_1\left[x_9^{(r-1)}\right] \oplus T_2\left[x_{14}^{(r-1)}\right] \oplus T_3\left[x_3^{(r-1)}\right] \oplus \mathbf{K}_1^{(r)}\\
\mathbf{x}_2^{(r)} &= T_0\left[x_8^{(r-1)}\right] \oplus T_1\left[x_{13}^{(r-1)}\right] \oplus T_2\left[x_2^{(r-1)}\right] \oplus T_3\left[x_7^{(r-1)}\right] \oplus \mathbf{K}_2^{(r)}\\
\mathbf{x}_3^{(r)} &= T_0\left[x_{12}^{(r-1)}\right] \oplus T_1\left[x_1^{(r-1)}\right] \oplus T_2\left[x_6^{(r-1)}\right] \oplus T_3\left[x_{11}^{(r-1)}\right] \oplus \mathbf{K}_3^{(r)}.
\end{aligned}$$

Here, T_0, T_1, T_2, T_3 are four lookup tables with 1 byte input and 1 word output and $\mathbf{K}_i^{(r)} = (K_{4i}^{(r)}, K_{4i+1}^{(r)}, K_{4i+2}^{(r)}, K_{4i+3}^{(r)})$ is the i-th word of the

[4]The number of rounds equals 10, 12 or 14, when n is 16, 24 or 32, respectively. Here, the number of rounds is fixed to 10.

[5]Note the last round is different in the sense that there is no `MixColumns`.

lsb	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
msb																
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
10	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
11	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
12	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
13	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
14	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
15	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 4: SBox S.

lsb	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
msb																
0	c6	f8	ee	f6	ff	d6	de	91	60	02	ce	56	e7	b5	4d	ec
1	8f	1f	89	fa	ef	b2	8e	fb	41	b3	5f	45	23	53	e4	9b
2	75	e1	3d	4c	6c	7e	f5	83	68	51	d1	f9	e2	ab	62	2a
3	08	95	46	9d	30	37	0a	2f	0e	24	1b	df	cd	4e	7f	ea
4	12	1d	58	34	36	dc	b4	5b	a4	76	b7	7d	52	dd	5e	13
5	a6	b9	00	c1	40	e3	79	b6	d4	8d	67	72	94	98	b0	85
6	bb	c5	4f	ed	86	9a	66	11	8a	e9	04	fe	a0	78	25	4b
7	a2	5d	80	05	3f	21	70	f1	63	77	af	42	20	e5	fd	bf
8	81	18	26	c3	be	35	88	2e	93	55	fc	7a	c8	ba	32	e6
9	c0	19	9e	a3	44	54	3b	0b	8c	c7	6b	28	a7	bc	16	ad
10	db	64	74	14	92	0c	48	b8	9f	bd	43	c4	39	31	d3	f2
11	d5	8b	6e	da	01	b1	9c	49	d8	ac	f3	cf	ca	f4	47	10
12	6f	f0	4a	5c	38	57	73	97	cb	a1	e8	3e	96	61	0d	0f
13	e0	7c	71	cc	90	06	f7	1c	c2	6a	ae	69	17	99	3a	27
14	d9	eb	2b	22	d2	a9	07	33	2d	3c	15	c9	87	aa	50	a5
15	03	59	09	1a	65	d7	84	d0	82	29	5a	1e	7b	a8	6d	2c

Figure 5: SBox S'.

r-th round key. Note that the last round uses another lookup table T_4 (later mentioned as SBox Table 4).

The SBox Tables T_i are generated from two constant 256-byte tables, designated as S and S'. Figures 4 and 5 display the tables as 16-by-16 byte matrices.

The SBox tables are defined by the following concatenations:

$$\begin{aligned}
T_0 &= (S', S, S, S \oplus S'), \\
T_1 &= (S \oplus S', S', S, S), \\
T_2 &= (S, S \oplus S', S', S), \\
T_3 &= (S, S, S \oplus S', S'), \\
T_4 &= (S, S, S, S).
\end{aligned}$$

These 5 precomputed lookup tables provide a noticeable increase of performance for encryption[6], since they enable each round to be composed solely by lookups and bitwise additions. The obvious drawback is the memory consumption of those tables (10 KB).

The memory / execution time trade-off can be adjusted to various degree of precomputations: from no precomputation to two 256-byte tables or two 2048-byte tables or even ten 1024-byte tables. The performance is generally proportional to the size of the precomputations.

I.3. RSA

In 1978, R. Rivest, A. Shamir and L. Adleman described an algorithm for public-key encryption [98]. The algorithm was patented in 1983 [97] and the patent expired in September 2000. The security of RSA is based on the intractability of the integer factorization problem.

I.3.1. Key generation

Two large and random prime integers p and q are chosen. Their product is called the *modulus*: $n = p \cdot q$. Another integer is computed: $\phi = (p-1) \cdot (q-1)$. We select a random number e, with $1 < e < \phi$ and $\gcd(e, \phi) = 1$. By the extended Euclidean algorithm, we find the unique integer d such as $1 < d < \phi$ and $e \cdot d \equiv 1 (\bmod \phi)$. The public key is (n, e) and the private key is d.

There are constraints on the choice of the parameters to be safe from common attacks (refer to [69]).

[6]For decryption, 5 other lookup tables are defined.

I.3.2. Encryption - Decryption

We define two parties $\mathcal{A}$ and $\mathcal{B}$ that want to communicate. In order for $\mathcal{B}$ to encrypt a message m $(0 < m < n)$ to the attention of $\mathcal{A}$, $\mathcal{B}$ obtains $\mathcal{A}$'s public key and computes the ciphertext $c = m^e \bmod n$. $\mathcal{B}$ finally sends c to $\mathcal{A}$.

$\mathcal{A}$ decrypts $\mathcal{B}$'s ciphertext with her private key d by computing $m = c^d \bmod n$.

Anyone can use $\mathcal{A}$'s public key to encrypt a message, but $\mathcal{A}$ is the only one that can decrypt it, because she has the private key.

I.3.3. Chinese Remainder Theorem

The Chinese Remainder Theorem (CRT) facilitates the modular exponentiation for the decryption (refer to [52, 93]). Let us define

$$d_p = d \bmod (p-1) \text{ and } d_q = d \bmod (q-1).$$

Then we can compute,

$$S_p = (c \bmod p)^{d_p} \bmod p$$

and

$$S_q = (c \bmod q)^{d_q} \bmod q.$$

S_p and S_q are combined to find c. There are several techniques to achieve the recombination: for example,

$$c = \left[\left(S_p \cdot q \cdot \left(q^{-1} \bmod p\right) + S_q \cdot p \cdot \left(p^{-1} \bmod q\right)\right)\right] \text{ (Gauss's)}$$

or

$$c = S_q + \left[(S_p - S_q) \cdot \left(q^{-1} \bmod p\right) \bmod p\right] \cdot q \text{ (Garner's).}$$

For security-related issues in RSA (cf. to [69]), the CRT usually deals with operands having half of the bit length of normal RSA operands. Therefore its application increases the performances by 4 with lighter hardware[7]. However, extra hardware is required to reduce the inputs and recombine the outputs. Nevertheless numerous applications integrate those operations and have recourse to CRT, because of its overall advantages.

The recombination algorithm and its implementation must be carefully reviewed in order to avoid attacks (see [12] for an example).

[7]In practical implementation, the actual increase factor is near to 3.5 because precomputations and final recombination. The increase factor can reach 8 if the two modular exponentiations are performed in parallel, with the expense of doubling the hardware.

```
$z := 1$
for $i := t - i$ to $0$
    $z := z \cdot z \bmod n$
    if $(s_i = 1)$
        $z := z \cdot m \bmod n$
$c := z$
```

Figure 6: Modular exponentiation scanning the exponent from MSB to LSB (left to right). The advantage is that only one register is required.

```
$z := 1$
$y := m$
for $i := 0$ to $t - 1$
    if $(s_i = 1)$
        $z := z \cdot y \bmod n$
    $y := y \cdot y \bmod n$
$c := z$
```

Figure 7: Modular exponentiation scanning the exponent from LSB to MSB (right to left). The advantage is that the square and the multiplication operations can be performed in parallel.

I.3.4. Modular exponentiation

Let us briefly review the way that the modular exponentiation is performed and especially on software implementations. The modular exponentiation is usually decomposed in a succession of modular multiplications. The basic technique is the *square and multiply*: scanning through the bits of the exponent, the temporary value is squared each round and multiplied only if the considered bit is 1. The exponent can be examined from MSB to LSB, or vice versa, producing slightly different algorithms. Let us suppose we want to compute $c = m^e \bmod n$, with the binary decomposition of c being $c = \sum_{i=0}^{t-1} c_i \cdot 2^i$. Figures 6 and 7 display such algorithms.

These exponentiation techniques are vulnerable to attacks exploiting the time difference induced by the conditional multiplications. The condition being directly related to the exponent (private or public key in RSA), the overall execution time provides information on the exponent (see [29, 53, 103]). In order to prevent someone from exploiting the execution time to deduce RSA keys, techniques have been proposed to mask the leakages by eliminating the conditions (*Square-and-multiply-always*) or by a more efficient way (for example, see [22]).

Finally, let us mention sliding window techniques: instead of scanning the exponent bit per bit, several bits are considered. In the left to right exponentiation, the multiplication becomes $z := z \cdot (v \cdot m) \bmod n$, according to the value v of those bits. It increases the computation speed if all possible $v \cdot m \pmod m$ values are precomputed.

I.4. Summary

We presented two cryptographic algorithms which are the most generally used in current applications. RSA is an asymmetric scheme and, among other applications, it is used to transmit an encrypted data over an insecure channel. This is performed by encrypting a plaintext with the public key of the receiver. Only the receiver can decrypt the encrypted data (the ciphertext) with the private key, back into the plaintext. Usually, asymmetric schemes are used to securely transmit the secret key for a further communication using a symmetric scheme, for example AES. Because of their structure, symmetric crypto-schemes are usually faster than asymmetric ones.

Further readings

[25] *Joan Daemen and Vincent Rijmen.* The design of Rijndael, AES - The Advanced Encryption Standard. *Information Security and Cryptology. Springer, 2001.*

[30] *Network of Excellence in Cryptology ECRYPT. AES Lounge. Available online at* `http://www.iaik.tugraz.at/research/krypto/AES/`*.*

[35] *Oded Goldreich.* Foundations of Cryptography: Volume 2, Basic Applications. *Cambridge University Press, New York, NY, USA, 2004.*

[69] *Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone.* Handbook of applied cryptography. *CRC Press, Boca Raton, Florida, 1996. Available online at* `http://cacr.math.uwaterloo.ca/hac`*.*

[79] *OpenSSL the open-source toolkit for SSL / TLS. Available online at* `http://www.openssl.org/`*.*

CHAPTER II

Cache Architecture

Abstract: Moore's law predicts that the complexity of an integrated circuit will double every 18 months. For more than two decades now, this empirical observation has not only correctly expressed the continuous progress of technology but has also served as a goal for the entire silicon industry. The transistors, the elementary parts of a chip are becoming smaller and faster, enabling the chips to integrate more transistors on an equivalent area and to run at higher frequencies.
However not all silicon-related technologies increase their capacity and speed according to Moore's law. The memory speed for example is only improved by a little percentage a year, at best. Nevertheless, as the memory is a central part in most of the chips and as memory is usually slower than its data-path, it is crucial to smartly interface the memory with the chip, with the smallest impact on the chip's performances.
A generally adopted solution is the cache. This Chapter introduces the cache architecture and details the decisive role it plays in modern processors.

II.1. Microprocessor

A processor (or microprocessor) is a complex circuitry of millions of transistors. It plays the central role of the authority from where all aspects of the computer's behavior flow out. Nowadays a microprocessor is composed of a variety of sophisticated elements (power management unit, IO, etc.) helping the *Central Processing Unit* (CPU) to work in an optimal way. The CPU is generally made out of several components:

- *Arithmetic and Logic Unit* (ALU) performing elementary arithmetic and logic operations;
- *Control Unit* (CU) executing the instructions, with the help of ALU if needed;
- *Registers* which are small memories capable of being manipulated at the CPU's frequency. The most common ones are the *Program Counter* (holding the address of the instruction being executed), the *Accumulator* (holding the data currently used), ... and registers available for computations and status;

- *Clock* synchronizing all components of the CPU;
- *Input Output Unit* dealing with the memory and enabling the CPU to communicate with its peripherals. In particular the *Memory Management Unit* (MMU) handles memory accesses for the CPU.

Moreover, nowadays processors integrate complex units:

- *Floating Point Unit* (FPU) capable to perform arithmetic operations on real numbers represented in floating point;
- multiple ALUs working in parallel or independently, in order to achieve more operations per clock cycle;
- high level of pipeline that segments the instruction process in *micro-operations* (μops) (*e.g.* resp. 5, 10 and 20 levels for Pentium, Pentium Pro/II/!!! and Pentium 4), enabling higher work frequencies;
- *Branch Prediction Unit* that predicts the result of a conditional test before computing the condition, this enables the pipeline not to stall; and the cache, the main topic of this chapter.

Now that its components have been depicted, let us define its purpose. The microprocessor is in charge of executing programs: operating systems, applications, and so on. Zillions of instructions are sent to and analyzed by the processor in order to perform tasks. As a large percentage of general programs are `load` and `store` instructions (20 to 40% for `load` as reported in [87] due to the limited number of registers composing the CPU), it is clear that high speed communications with memory are of the greatest importance. The memory hierarchy of a computer includes at one end the registers with small access time (in nano- or picoseconds) to removable media (optical and magnetic disks) at the other end with access time in seconds. The whole motivation of cache appears when considering this speed discrepancy: CPU needs to access data at its working frequency (usually higher than 1 GHz for a PC); however for a memory, the faster and the larger, the slower and the more expensive[1]. Over the past decade, the CPU speed increased by more than ten times, while the memory speed only doubled.

To buffer the difference between processing and memory speeds and stay balanced between performance and cost, the microprocessor integrates a small but fast memory close to the CPU, hiding the main memory's latency. This memory is called the CPU cache or simply the cache. Its utopian role is to contain the next data to be accessed. It is indeed extremely difficult to predict the future needs of a program, essentially because they often depend on the intermediate results of the program itself and the external stimuli.

[1]It is generally admitted that the manufacturing price of a dice is proportional to the square of the size.

There is a distinction among memories with respect to *volatility*: *i.e.* their ability to hold their data without power. Non-volatile memories (principally Flash but also EEPROM, etc.) are used in cell phones, portable media players and USB storage drives. However non-volatile memories are slower than volatile memories. Therefore volatile memories are used for main memory in PCs, where the lost of their content when powered off is acceptable. They are usually divided in two families [94]. The first kind is the *Static Random Access Memory* (SRAM). One bit and its inverse are stored by two cross-coupled inverters and accessed by two control transistors that, when opened, output the stored bit and its inverse to the bit lines (BL and $\overline{\text{BL}}$). This symmetry provides better margins and therefore faster accesses. These memory cells are called static since they do not require refresh, in contrast to *Dynamic Random Access Memory* (DRAM). The latter stores a bit of information of a capacitance and the access to read or write that bit is controlled by a single transistor. As the capacitance leaks electrons, the stored information fades away unless it is periodically refreshed. The simplicity of DRAM allows very cost-efficient and denser designs.

The utilization of SRAM is then focused for high-speed and limited size applications, *e.g* CPU registers and caches. In contrast, DRAM are used for higher density, high capacity and low price per bit applications like PC or graphic RAM (DRAM, SDRAM, DDR, QDR, etc.).

II.2. Hiding the main memory's latency

Whenever a data is required, the CPU first looks inside the cache. If the cache holds the data, a *cache hit* occurs and the data is sent to the CPU. Otherwise the data has to be fetched from the main memory and brought back to the CPU. This is a *cache miss*. The spacial and temporal localities motivate to bring into the cache not only the requested data but also its surrounding ones, for probable future accesses. The initiation of memory transfer consumes more time than the actual transfer. However, the surrounding data is now in the cache and the probability of cache hits in the near future is increased.

Mark Hill provided in [40] a model of cache misses dividing them into three categories. They are known as the 3Cs:

- *Compulsory* misses that occur on the first reference of a data;
- *Capacity* misses that are caused by the limited size of the cache;
- *Conflict* misses that take place because the data has been evicted earlier.

II.3. Principle of locality

According to the work of late 1960's researchers at IBM, nearly all programs are extremely repetitive in nature. [87] reports 90% of the execution time is spent over 10 to 15% of the code. If a loop can be kept in cache, it could be efficiently executed by the CPU and only the non-repetitive parts would require the CPU to access the main memory. The principle of locality [28] details this phenomenon under two aspects:

- *Locality in space* (or *locality of reference* or *spacial locality*) states that most programs execute repetitively over a small area (parts of code or data): the next reference to be accessed in a near future is likely to be in vicinity of the one just accessed;
- *Locality in time* (or *time of reference* or *temporal locality*) expresses the fact that a reference that has been accessed is likely to be accessed again in a near future.

II.4. Model of memory hierarchy

For the sake of clarity, we first introduce a simple memory hierarchy, comprising a cache and a main memory (Figure 1) in a Hardware architecture[2]. The CPU shares an address bus and a data bus with the memory. The cache intercepts the access requests in order to check if it holds the requested data. If it does, a cache hit occurs and the cache bypasses the main memory and provides the data to the CPU. Otherwise the main memory provides the data and its surrounding blocks; the CPU receives the data and the cache stores the whole block as a cache line.

We follow the notations of [74]. We use byte addressable memory, *i.e.* each address refers to a byte. Let 2^c be the byte size of the cache and 2^m the byte size of the main memory[3]. The unit used for transfers inside the memory hierarchy is called a block or a *cache line*. The *cache line size*, *i.e.* the number of bytes inside a cache line, is usually chosen as a power of 2, say 2^o. Let 2^l be the number of cache lines. Therefore $2^c = 2^l \times 2^o$ (and $c = l + o$). Also, let a be a main memory address: $a = \langle a\,[m-1:0]\rangle$.

II.5. Fully associative cache

In a fully associative cache, a block can be placed in any cache line. In this case, the main memory address a is split into two parts: the

[2]See [87] for description and distinction between Harvard and von Neumann and the impact on cache architecture.

[3]Note that m equals 32 for most 32-bit processors. For 64-bit processors, m will tend to be 64 but it stands between 32 and 64 for current architectures.

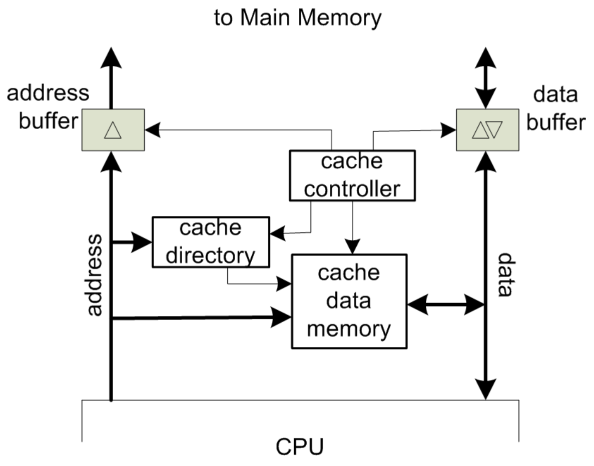


Figure 1: Schematic view of a memory hierarchy.

tag $\langle a\,[m-1:o]\rangle$ and the *offset* $\langle a\,[o-1:0]\rangle$ (Figure 2). We have seen above that in case of a cache miss, a whole cache line is transfered from the main memory to the cache. Let C be the data of the cache line. The addresses of elements of C share the same tag. Therefore, the cache stores the tag $\langle a\,[m-1:o]\rangle$ and the data C (Figure 3).

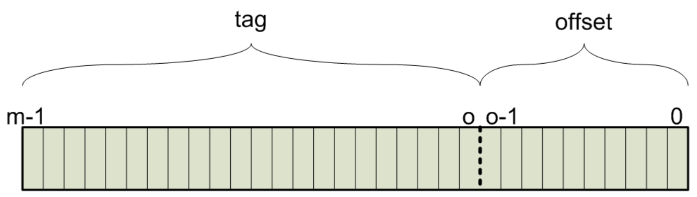


Figure 2: Composition of a main memory address for fully associative cache.

On a CPU request of a particular data, the MSB of its address is compared with the tag of each of the 2^l cache lines. The comparisons have to take place at the same time, forcing the hardware to comprise 2^l comparators, *i.e.* one per cache line. Even if a comparator can be done by specialized 7-transistors RAM, the hardware penalty is substantial for fully associative cache.

II.6. Direct-mapped cache

Expensive parallel comparisons can be avoided by using a direct-mapped cache. Its principle is completely different. Each memory location has

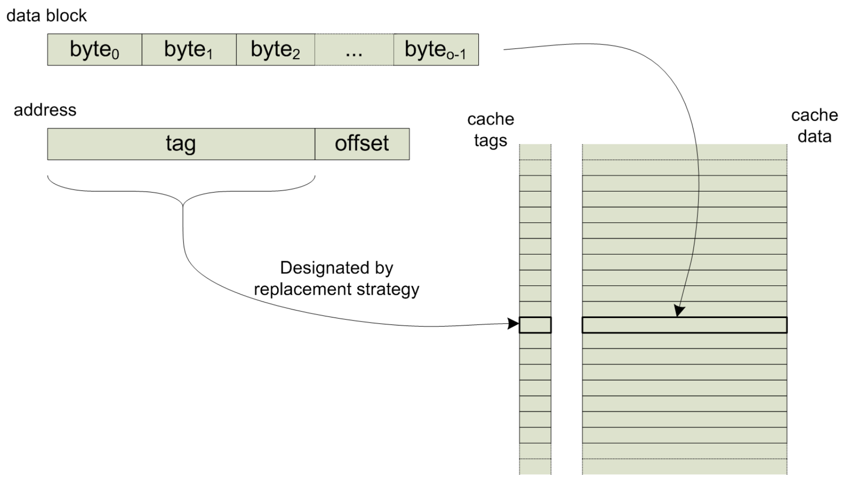


Figure 3: In case of cache miss, the cache line is fetched from the main memory and the replacement policy chooses the cache line to be evicted. The corresponding tag is also updated by the new address's MSB.

only one possible location in cache: the cache address ca is deduced by the LSB of a (see Figure 4):

$$ca = \langle ca\,[c-1:0]\rangle = \langle a\,[m-1:0]\rangle \bmod 2^c = \langle a\,[c-1:0]\rangle$$

In this scheme, it is easy to answer the CPU's data request at address b, since there is only one location in the cache to look at. The cache must simply check if the tag of cache line $\langle b\,[c-1:o]\rangle$ is $\langle b\,[m-1:c]\rangle$.

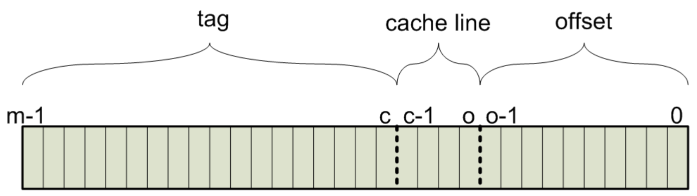


Figure 4: Composition of a main memory address for direct-mapped cache.

The drawback of direct mapped caches is that they behave inefficiently in case of collisions. However their access and replacement policy is considerably simpler than with fully associative caches.

II.7. Set associative cache

This hybrid cache tends to conciliate the advantages of direct-mapped and fully associative caches. It provides a set of possible locations for a cache line. Contrarily to fully associative caches, the set is limited to a small integer k, usually chosen as a power of 2. The cache is then referred as k-way set associative. It can be seen as if the cache is divided into k smaller caches, each one of them being direct-mapped. One can pinpoint that when $k = 1$ the cache is direct mapped and when $k = 2^l$ the cache is fully associative.

The set is given by $\langle a\left[\log_2(k) + o - 1 : o\right]\rangle$ (refer to Figure 5).

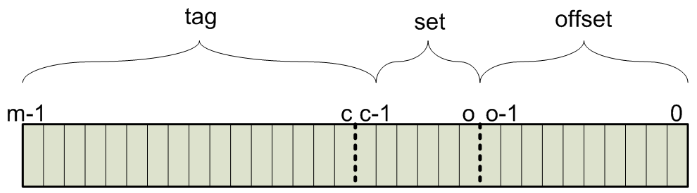


Figure 5: Composition of a main memory address for k-way set associative cache.

Now, the tag must be compared with k locations at the same time. The drawback in the hardware is therefore significantly lower than in the fully associative cache. Moreover, the risk of collision is limited. As a result, set associative caches are often used as a compromise for CPU caches. Table 1 summarizes the guideline for selecting a cache type following the system's requirements.

	Simplicity	Hit-rate
Fully associative	$--$	$++$
Set associative	$-$	$+$
Direct-mapped	$+$	$--$

Table 1: Condensed table of pros and cons in the selection of cache type.

II.8. Replacement policy

As it has a limited capacity, the cache ends up by being full: each new entry (successive to a cache miss) evicts previously stored data. The choice of the victim data to be replaced is given by the cache's replacement policy (also called cache algorithm) [5]. In fully and set- associative caches, it designates which ways must be evicted. The criteria to choose the appropriate replacement policy consider the required number of extra bits, the number of operations needed in the case of cache-hit or miss and on the expected performance. The most frequent ones are:

- *Least Recently Used* (LRU): To keep track of the least recently used location, the cache must store the order of the accesses from the most to the least recent one. If there are N possible locations for a data, there are $N!$ different ways to arrange those locations. Therefore this cache algorithm gets hardware consuming with high N: $\lceil \log_2 N! \rceil$ bits are needed to store the order, *i.e.* 0 for $N = 1$, 1 for $N = 2$, 5 for $N = 4$, 16 for $N = 8$, and so forth. Moreover, whether it is a hit or a miss, updating those order bits requires three operations (read, update, write); this is generally incompatible with the high speed requirements of cache-hits.
- *Pseudo-LRU* or Tree-LRU algorithm approximates the LRU but improves upon it in several ways: it consumes half of the memory bits and only a write operation is needed in case of cache hit. However, if a miss occurs, the replacement requires three operations as with LRU. This is not a too big penalty since the replacement itself is slow anyway. Refer to [38] for more details.
- *Least Frequently Used* (FRQ): A pointer keeps track of the least frequently used location. At every cache access, if the pointer refers to the accessed data, the pointer is randomly updated to point to another location. Therefore, after some accesses, it is generally pointing to a least frequently used location. In case of replacement, the pointer is also updated to avoid direct replacement. Read and write operations are needed on each access. However it does not need as many bits as LRU.
- *Random replacement* allows simple designs but cannot guaranty direct replacement of the last accessed location.
- *Not Last Used* (NLU), works as the previous strategy, except it keeps a pointer to the most recently used location to avoid direct replacement. It only spends a write operation and consumes $\lceil \log_2 N \rceil$ memory bits.
- *Round Robin* identifies the next line to be replaced on a first-in-first-out basis (FIFO) [124].
- *Belady's Min* at last is a theoretical strategy. It discards the location not needed for the longest time, which is then the best choice of replacement location. This ideal case can not be implemented in processors.

II.9. Specialized caches

Even if the basics of cache principles are fairly easy to understand, modern microprocessors integrate various types of cache in several parts of the design. The caches always hide memory's high latency by storing elements close to its target (*i.e.* the CPU in this case, but caches can serve many other causes). Those elements can be data but also instructions. A pipeline with separate data and instruction caches is called *Harvard* in reference to the former architecture with distinct data and program memories. Microprocessor's designs appear to be fairly intricate and the control logic becomes very complex. However it enables the hit rates to increase and the whole processor to be more efficient. Consider for example that the hit rate is improved from 96% to 98%. The benefit of 2% seems negligeable but it means that the main memory requirements for the cache drops from 4% to 2%. It represents a solid improvement on the required bandwidth. This explains the motivation in complex but efficient cache architectures.

Multi-levels cache is the widely adopted solution to deal with the trade-off between latency and hit rate. Remember the larger the cache, the greater the hit rate. However larger caches are slower. Therefore, the small caches – level 1 – close to the CPU are backed up by larger ones – level 2 – with better hit rates. High-end x86 servers' processors even integrate a third level cache of several megabytes (from 2 to 256). They usually cannot be physically placed onto the microprocessor's chip (*on-core*) and are then inserted either inside the packaging (*on-chip*) or outside (*off-chip*).

Victim cache (or Victim Buffer) holds the data that have just been evicted. It is inserted between the main cache and the refill path. Since the introduction of its *Thunderbird* processor, AMD integrates a small and fast victim buffer (*e.g.* 8 entries) to improve the performance of the cache architecture. Whereas Intel designs an inclusive relationship between L1 and L2 caches, AMD chooses the exclusive option: a data is at most in one level of cache. This strategy makes the total usable cache size bigger than in an inclusive scenario, but limits the performances of the L2 cache [27].

Virtual Memory or address translation is a common technique for the microprocessor to provide memory isolation. The program's code and data are isolated from other ones in their own independent simplified memory space. The processor needs to translate the physical addresses on the main memory to the virtual addresses; this is done by the MMU. For latency matters, the translations are stored in *Translation Lookaside Buffer* (TLB): a content addressable memory in which the input is the virtual address and the search result is the physical address. A cache can be tagged either by virtual or physical addresses. Most L1 caches are virtually tagged as they bypass the TLB or MMU and therefore are faster. However virtual memory suffers from potential alias problems: different virtual addresses might represent a unique physical memory and a modification through one virtual address might then corrupt the value referred by the others. The cost related to aliases grows with the cache's size and so L2 are generally physically indexed.

Trace cache is inserted to increase the instruction fetch bandwidth. This mechanism, proposed in [99], stores traces of instructions that have already been fetched. It enables the CPU to avoid interferences with the branch prediction unit. It is also used to store translations of complex x86 instructions that have already been decoded.

II.10. Examples

Let us illustrate a cache design by examining the memory hierarchy in two examples of general-purpose microprocessors and an example of 32-bit microcontroller.

II.10.1. K8

The K8 microarchitecture of AMD is the base for the Athlon 64, Athlon 64 FX and Opteron processors [100, 130]. Refer to Figure 6.

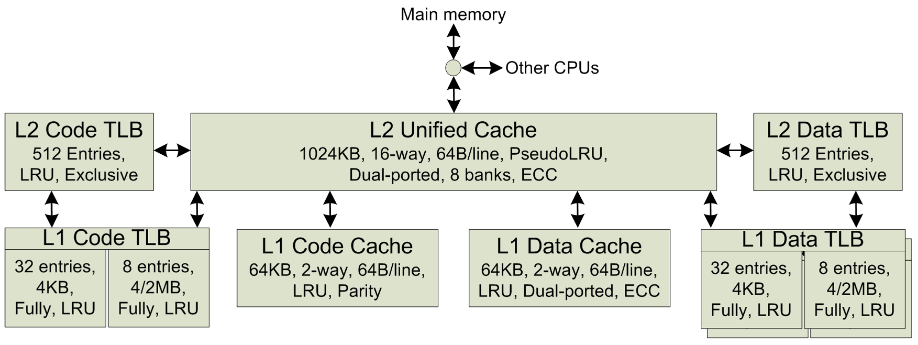

Main memory
Other CPUs
L2 Code TLB
512 Entries, LRU, Exclusive
L2 Unified Cache
1024KB, 16-way, 64B/line, PseudoLRU, Dual-ported, 8 banks, ECC
L2 Data TLB
512 Entries, LRU, Exclusive
L1 Code TLB
32 entries, 4KB, Fully, LRU
8 entries, 4/2MB, Fully, LRU
L1 Code Cache
64KB, 2-way, 64B/line, LRU, Parity
L1 Data Cache
64KB, 2-way, 64B/line, LRU, Dual-ported, ECC
L1 Data TLB
32 entries, 4KB, Fully, LRU
8 entries, 4/2MB, Fully, LRU

Figure 6: Memory Hierarchy of the K8 of AMD (cache size may vary regarding the technology).

The L2 cache is unified, *i.e.* underneath TBL and L1 caches are filled from L2 either for code or data. It is also exclusive in the sense that any 8-byte data can only be either in L1 (code or data) or in L2.

The next level comprises 4 parts:

- The L1 code cache fetches 16 bytes per cycle from L2. Each byte is stored in a group of 10 bits rather than 8, along the additional bits marking the boundaries of the instructions. Parity protection is redundant enough as L2 incorporates Error-Correcting Code (ECC).
- The code TLB keeps copies of the translation from virtual to physical addresses. The TBL is split into sections: one mapping page table entries to 4KB and the other one to 2 or 4 MB[4].

[4]The Operating System (OS) maps different sections of the virtual address space with different sizes.

- The data TLB holds two identical copies to allow two data access' translations per cycle. As the code TLB, it is split into two sections.
- The data cache has 64 bytes per line. It is split into 8 banks (multibank cache) of 8 KB and can fetch two 8-byte data per cycle, if the data are in different banks (no bank conflict). The 6-bit offset is divided into two 3-bit parts representing the bank and the byte. A bank conflict occurs if the two requests have the same bank and different sets (also called indexes in [3]). A closer look is given in Figure 7.

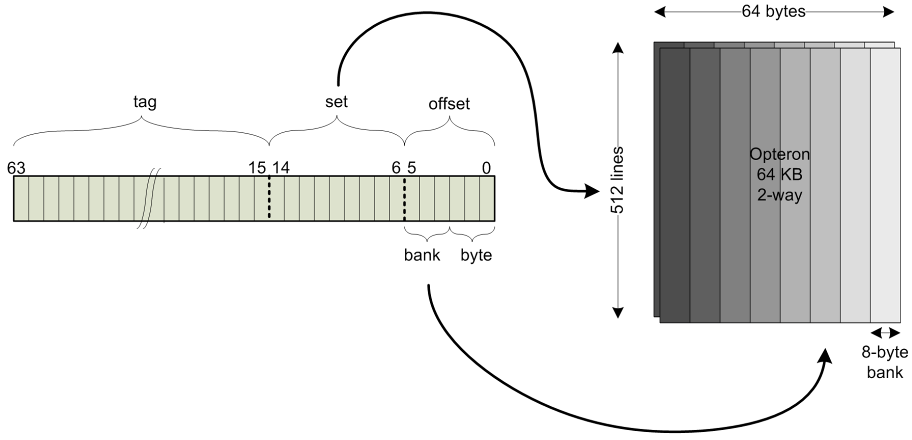


Figure 7: Structure of the L1 data cache of K8 processors.

Figure 8 is a picture of the die of a K8. It is worth seeing the proportion of the chip dedicated to memory.

Moreover, the memory hierarchy is shared in different specific architectures. The sharing can be at the main memory level where several processors form a tightly coupled network by sharing a system bus to the main memory. It is the common configuration for multiprocessor systems. On another side, the sharing can be as intimate as up to the L1 data cache. Hyper-Threading Technology of Intel is a good example. It consists in duplicating the architectural state of the processor, but not the execution units, in order to be equivalent to the OS as two processors. Therefore, two threads can be simultaneously executed. For example, unoccupied execution units can be used by the second thread when the first one stalls. In this context, it is clear that data coherency must be carefully controlled. This topic is not explored here. For more details, refer to [38, 106].

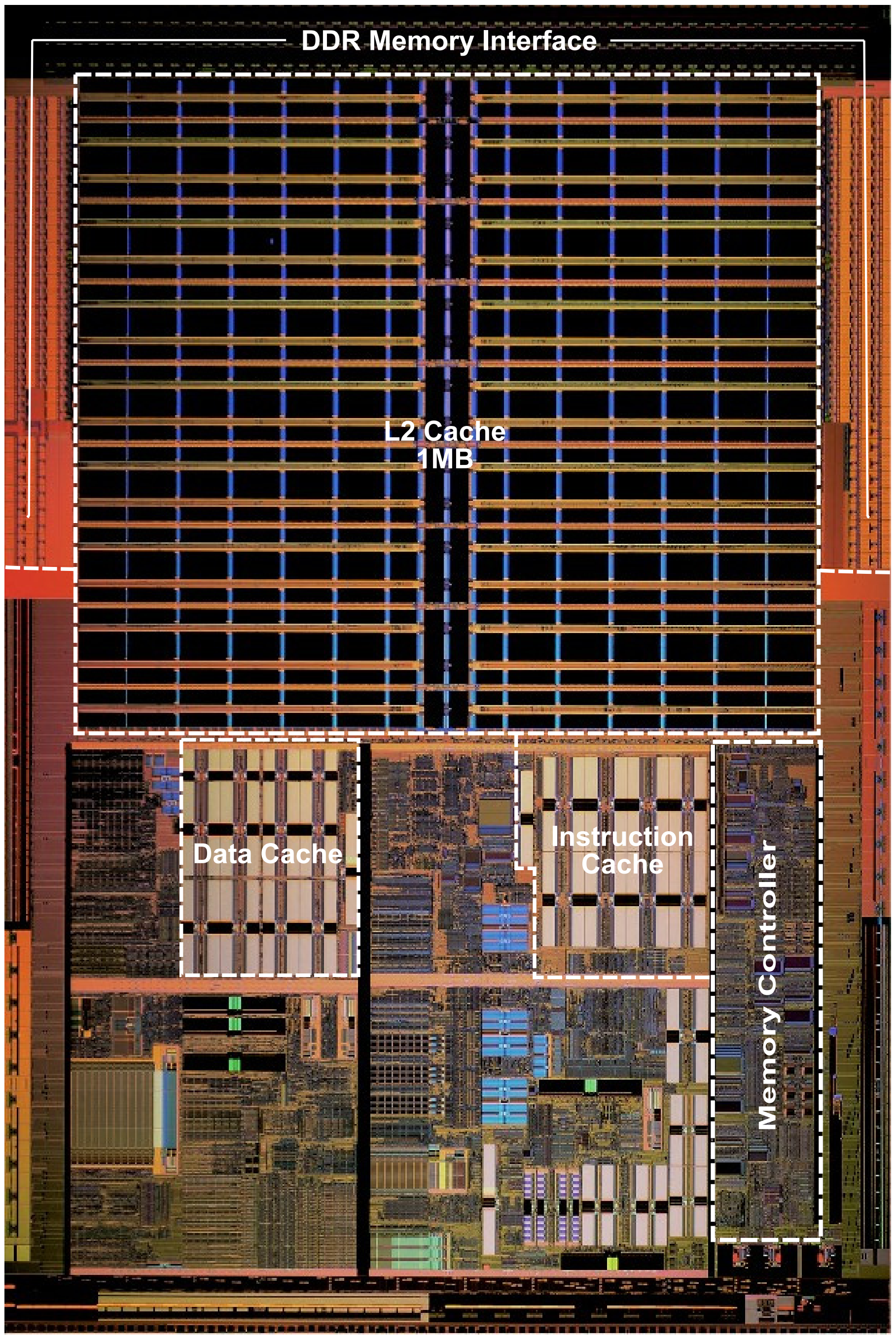


Figure 8: Die of a AMD K8 with 1MB unified L2 cache (0.13μ). Observe the proportion of the processor dedicated to memory (Source: AMD Corp.).

II.10.2. Intel's P4

The Pentium 4 microarchitecture of Intel is based on NetBurst microarchitecture. Figure 9 shows the memory subsystem of a 0.09μ Prescott Pentium 4 [41, 101, 106].

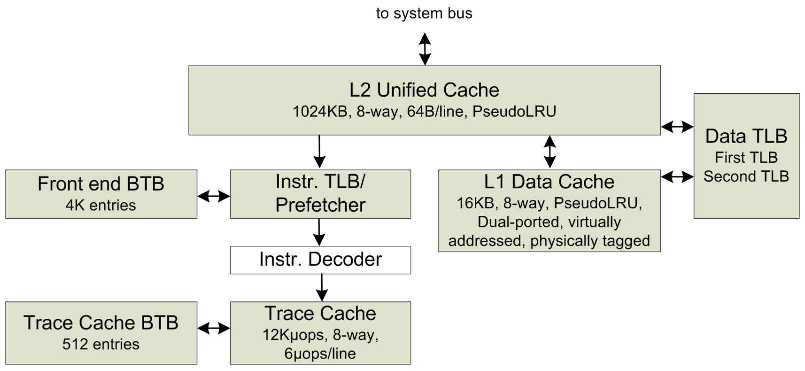


Figure 9: Memory Hierarchy of the P4 of Intel (cache size may vary regarding the technology).

The L1 instruction cache differs from other architectures. As mentioned earlier, the *Execution Trace Cache* (or Trace Cache) already stores decoded IA-32 instructions and μops. It delivers up to three μops per cycle to the out-of-order execution logic. Its hit rate is claimed to be similar to an 8KB or 16KB conventional instruction cache. The Instruction TLB and Front End Branch Table Buffer (BTB) steer the front end in case of cache miss of the trace cache.

The data part is composed of a classic L1 data cache, with dual-port and 8-way, and a proprietary TLB system [125]. The latter has two TLB: a first one with truncated address translations for fast accesses and the second one with accurate translations. The first TLB quickly identifies a tentative way in the data cache; the data is read out and its tag is compared with the information of the second TLB. The data is the requested one if the tentative and validated physical addresses match.

II.10.3. MIPS32 core

First let us add a quick note on the definition of a processor. There is a wide variety of silicon chips that manipulate data and produce structured results (microprocessor, microcontroller, Digital Signal Processor (DSP), etc.). In particular, microcontroller is generally admitted as the term used for a computer-on-a-chip that controls electric devises. It is usually a stand-alone product with peculiar attention to cost- and energy-effectivenesses. A microprocessor however is embedded in more complex electronic designs and requires surrounding chips to perform and operate usefully. The distinction tends to get blurry as some micro-controllers integrate 32-bit ALU, caches and large memories and as they are able to support elaborated OS. Also, hand-help devices as smart cards, cellular phones, etc. require more processing capabilities for current applications (security, multimedia, etc.). However, current microprocessors still exhibit higher data processing functionalities and run at higher frequencies. Even if we focus on the general-purpose microprocessors in this dissertation, here is a short description of the MIPS32 4Kc's architecture to emphasize its cache functionalities (refer to [72]).

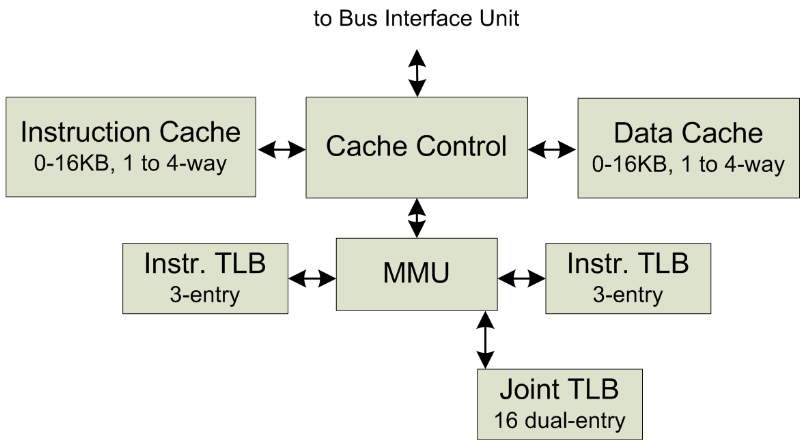


Figure 10: Memory Hierarchy of the 4Kc processor of MIPS.

The core implements a MMU that contains 3-entry instruction and data TLB (ITLB/DTLB) and a 16 dual-entry joint TLB (JTLB) with variable page sizes. The instruction and data cache are configurable in size (from 0 to 16KBytes) and in associativity (from direct-mapped to 4-way set associative). See Figure 10.

II.11. Summary

We gave the principles of cache mechanisms and architectures and explained the motivation of the processor industry to provide efficient caches to their CPU. We also showed that the cache takes advantage of the locality at the code and data's levels. The next Section discusses the particular problem of implementing a security program, especially a cryptographic application, onto a processor. For throughput and security issues, large precomputed tables and random accesses are often mandatory; and open several breaches.

Cache architectures and mechanisms are crucial aspects of microprocessors. The implementation and the integration of the cache require a careful analysis, because of its highly constrained role.

Further readings

[38] *Jim Handy.* The cache memory book (2nd ed.): the authoritative reference on cache design. *Academic Press, Inc., Orlando, FL, USA, 1998.*

[74] *Silvia M. Mueller and Wolfgang J. Paul.* Computer Architecture - Complexity and Correctness. *Springer-Verlag, 2000.*

[87] *David A. Patterson and John L. Hennessy.* Computer Architecture: A Quantitative Approach. *Morgan Kaufmann, 3rd edition edition, 2003.*

[106] *Tom Shanley.* The Unabridged Pentium 4 : IA32 Processor Genealogy. *Addison-Wesley Professional, 2004.*

CHAPTER III

From confinement to side channels

Abstract: Information issued form a device might be related to a secret kept inside. The analysis of this information can then disclose parts of this secret. These so-called *side channel attacks* can be based on the analysis of potentially any information: execution time, power or temperature profiles, acoustic or electromagnetic waves, etc. The cache mechanism can be the source of information leakages, as it is shared between processes. We refer to this special kind of attack as *cache-based side channel*.
Cache-based side channels take part of their origin from the covert channels in secured systems. Early works already studied how data leakages might take place in such systems, between services being executed on the same security system but unauthorized to communicate with each other. The memory architecture, and *a fortiori* the cache, have been pointed out as hardware features suitable for covert channel communications between the services. A preceding collusion is needed to pull together a communication protocol. However side channel analyses are possible on the cache, provided one service is a cryptographic application: *i.e.* the underlying crypto-algorithm is generally of public knowledge (by Kerkhoff's principle [48, 49]) and the second service has a sufficient knowledge on the expected behavior of the first service. No collusion is then necessary. Furthermore the leakages are unintended and in general due to unawareness of the cache –or other– vulnerabilities.
This Chapter surveys the relations between covert channels and cache-based side channels. Moreover it shows that, for side channel, no collusion is necessary and evidences that the secret has been compromised are hard to detect due to the intentional character of the information leakages.

In 1973, Bulter W. Lampson formulated the foundations of the *confinement problem* [55]. It states that a program is confined if it is able to keep its information from unintended people or program. It covers the private data the costumers will submit to a secure service, proprietary programs constituting the service or secret data of the service, and so on. Lampson provided several examples of mechanisms by which a service might attempt to violate the confinement constraints expected by the customer. In [58] Lipner identified them as *covert channels*, informally defined by the exploitation of not intended means of communication to transmit information (*e.g.* file locks, busy flags, branch prediction tables, execution time, etc.). It is theoretically possible to understand the covert channels by analyzing all the system's internal data and mechanisms. However, the analysis efforts grow with the system's complexity and detection might require prior knowledge.

Covert channels are opposed to *overt channels* that use the protected data objects of the system to transfer information (*e.g.* buffers, files, shared memories, thread signals). Overt channels are controlled by enforcing the access control policy of the system. The policy details the conditions under which data objects may be manipulated. One part of the security analysis of a system must be to verify the implementation and application of the stated access control policy.

Conversely, preventing covert channels is more elusive: one should identify each of them and then block them. In addition, a covert channel requires the collusion between an outer or lower authorization process and a subject authorized to (make possible to) signal or leak information. Driven by strong military and government security requirements, there has been a large amount of covert channel researches. Lampson differentiates the covert channels into two categories: *storage channels* and *timing channels*. In timing channels, the information is convoyed by the timing of events. The receiving process thus needs an access to an independent clock to measure the timings, whereas storage channels are exploitable without external time reference. In [59] however, Loepere uses a different classification: the covert channel can either be a *static channel* or a *time decaying channel*. In both cases, a storage capacity of the system is affected by one process and later sensed by a different one. The difference lies in the amount of time the information is retained in the storage capacity: it is indefinite for a static channel, whereas it is limited for time decaying process. However, Lampson's distinction is the most widely used in the literature, so it is the one we will elaborate on in the following.

To better understand those two types of covert channel, let us study a request example in the hard drive of a system [132]. Our imaginary hard drive contains adjacent cylinders 51 through 59. Two processes `high` and `low` are executed on the system, with respectively high and low clearance levels. Both processes have read- (but not write-) access to the hard drive. The purpose is that `high` transmits classified information to `low` by the system, through the hard drive. The two different covert channels can be exploited.

- Storage channel:
  (1) `low` requests a data from cylinder 55 and gives up the CPU as soon as the request is completed.

(2) `high` now runs and asks to seek to cylinder 53 to transmit a zero bit (or cylinder 57 to transmit a one bit).
(3) `low` issues a request to cylinder 52 and 58 and observes the order of completion of those requests. If `high` sent a zero bit, the arm of the hard drive would have been traveling in the direction decreasing the cylinder's numbers (due to supposedly known hard drive's arm optimization). Therefore the request to cylinder 52 would be completed the first one. Otherwise, the request to cylinder 58 will finish first, hence `high` sent a one bit.

- Timing channel: the third step is replaced by a single request of `low` to cylinder 58. If `high` sent a zero bit (cylinder 53), the completion of the request would be slower than the case where `high` sent a one bit (cylinder 57).

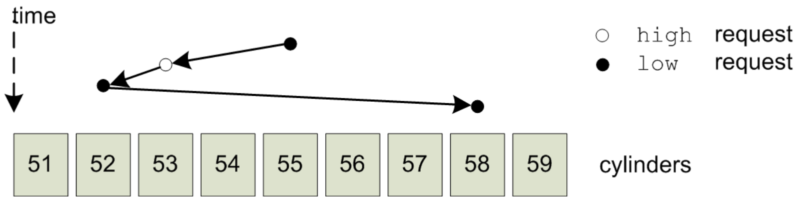


Figure 1: Illustration of the storage channel where `high` transmits a zero bit by seeking an access to cylinder 53. The arrow indicates the path the hard drive's arm will take to complete its requests.

Yet, collusion is not a mandatory condition for information leakages. Algorithms, implementation properties, physical emanations or carelessness are often the sources of the system's vulnerabilities and its adequate exploitation can expose protected data. In particular, civil cryptographic applications use well-defined cryptographic schemes[1] that permit the attacker to analyze the leakages by correlating his observations with the theoretical scheme. The attacks based on this principle are called side channels attacks. No collusion is then necessary. Moreover, *side channels* might acquire information in a stealthy way: the security system has no indication about the the theft of its secret. When covert channels can be detected afterward by considering the activities undertaken by the system (and puzzle out parts of the communication channels), detecting side channel activities is intrinsically more complex because of the unintentional character of the leakage.

The cryptographic community has known a vivid infatuation for side channel research since the preliminary work of P. Kocher [53]. Side channel analyses generally focus on hand-held embedded security devices, in

[1]According to Kerckhoffs' principle.

particular smart cards, because of their ubiquity[2]. Obtaining a bunch of cards of the same product family is fairly easy and cheap. But once a side channel leakage is identified and exploited on one single card, an attack on another card is immediate. Moreover, hand-held devices enable the attacker to get a close contact to the security chip: *e.g.* probing the data or power pins, or depackaging that permits access to near-field electromagnetic emanations of the card.

Until recently, side channel's impact was only focusing on embedded security device [29, 53, 54, 33] and had a limited influence to the PC world [13, 60, 119]. However, starting with the Trusted Computing effort [120] based upon the classical PC architecture, a new research vector is challenging those PC centric security efforts through side channel attacks [6, 7, 39, 56, 80, 81, 88, 121, 123].

Among those software PC vulnerabilities three recent papers [6, 81, 88] drew special public attentions and concerns. Beside overdone dissemination to media, the reason is that all three exploit an unavoidable element of nowadays high performance CPU architecture: the cache hierarchy. Nonetheless, none of them can claim the paternity of such *cache-based side channel attacks*. Actually [42, 108, 119] already pointed out how the observation of the cache behavior might be used as a potential channel, even if those papers were not targeting the analysis of existing cryptographic ciphers. Expanding on the idea of side channel through cache hit ratio (proposed by Kocher [53] and later by Kelsey *et al.* [47]), Dan Page provided a report [84] detailing a theoretical cache-based side channel against DES. Afterwards, although the first experimental cache-based side channel results against DES and AES were finally presented by [121], it was again Page [85] who explicitly described the foundations of the three recent publications [6, 81, 88]. Indeed, he characterized two different cache-based side channels: the *trace-driven* and the *time-driven* cache attack methodologies.

III.1. Time-driven methodology.

Time-driven attacks, as in [6, 39, 121], depend on the fact that when one runs the algorithm under attack, the execution time is directly affected by the number of cache misses. Using this relationship, the attacker can make algorithm-specific, statistically based inferences about the state during processing. Using these inferences and a large number of measurements that produce the desired feature, the attacker can relate the plaintexts to the key-related variables and hence uncover their value. They generally result in higher online workload since the attack

[2]Smart cards are currently used under various forms and for many applications all over the world. It is still limited in North America (because banking and communications were based on different systems) and in Africa.

is dependent upon statistical properties of the measurement in order to balance the generally poor signal to noise ratio.

III.2. Trace-driven methodology.

Trace-driven attacks, as in [42, 81, 88], rely on the ability of the attacker to capture a profile of cache activity that results from running the algorithm under attack. This requires that the attacker can get access to a profile in which the cache activity is observable and then process it to extract the cache activity profile from an other profile content. The result is that it records if the cache produced a hit or miss for every memory access. From such a trace it is relatively easy to relate S-box accesses for AES and DES assuming they are implemented via precomputed tables. By adapting the plaintext fed into the algorithm and hence provoking different cache access patterns, the attacker can uncover the values of the key dependent variables that cause this specific cache behavior and expressed in the trace profile.

Let us be more specific about these two kinds of attacks in the following Chapters. We give more details on the attacks and show how they can be improved.

Further readings

[53] *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.*

[55] *Butler W. Lampson. A note on the confinement problem.* Communications of the ACM*, 16(10):613–615, 1973.*

[58] *Steven B. Lipner. A comment on the confinement problem. In* SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 192–196, New York, NY, USA, 1975. ACM Press.*

CHAPTER IV

Time-driven attacks

Abstract: Time-driven cache-based attacks analyze the time difference in the execution of an algorithm over a processor, and deduces information about its inputs. The leakage –the time difference– takes place through the behavior of the cache architecture. It has the delicate mission to hold close the next needed data to the processor but it must be small enough to work at the processor's frequency. If the cache holds the data, it can provide them directly to the processor (cache hit); otherwise the data have to be fetched from memory, at large cycle expenses (cache miss). This task gets harder to achieve with cryptographic algorithms that generally need accesses to large precomputed data. And the security provided by the algorithm often relies on how random those accesses look like. Those particular conditions make the cache's job especially complex and might result in a higher miss rate, which translates to variable execution time. Therefore, as shown in this Chapter, the execution time analysis over AES encryptions can lead to –partial or full– recovery of the secret key.

IV.1. Preliminary note

Recall the notations introduced in the description of AES (Section I.2). Let us also define a function $time(x)$ measuring the time needed to complete an operation x. This is, however, the actual overall time to perform operation x including "machine" noise. Thus, two calls of this function can lead to different results because of the noise (multi-processing) and the fact that the encryption time might depend on the machine state before/after doing the operation x. Thus, $time(E_{AES}(\mathbf{p}, \mathbf{k}))$ denotes one particular measurement of the AES encryption time. Note also that we represent byte value in hexadecimal or binary basis by `x`$\cdot\cdot$ or `b`$\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot$, resp., with $\cdot$ is any hexadecimal or binary character: *e.g.* $165 =$ `x a5` $=$ `b 10100110`.

IV.2. Tsunoo *et al.*

The first practical time-driven attacks on cache were implemented in [121, 122] against various ciphers including MISTY1 [64], DES [77] and Triple-DES [77]. The cryptanalysis uses encryption times and infers parts of the expanded key since the encryption time depends on the cache activity (*e.g.* number of cache misses).

The authors moreover present two significant relations. In the simple model of an SBox accessed by the bitwise addition (XOR) of a known plaintext P_i and an unknown round key K_i, two SBox accesses ($P_0 \oplus K_0$ and $P_1 \oplus K_1$) are executed and their timing is measured. Whether their timing are similar or different, the attacker can use one of the following relations in order to infer bits of the key difference $K_0 \oplus K_1$:

$$P_0 \oplus K_0 = P_1 \oplus K_1 \rightarrow P_0 \oplus P_1 = K_0 \oplus K_1$$
$$P_0 \oplus K_0 \neq P_1 \oplus K_1 \rightarrow P_0 \oplus P_1 \neq K_0 \oplus K_1$$

They are respectively called the *non-elimination* and *elimination table methods* as they permit the attacker to partially deduce the key bits or eliminate impossible key values, resp.

They launched their elimination table attack upon DES under a Pentium III and had a success rate greater than 90% with 2^{23} known plaintexts and 2^{24} calculations.

Using those results, they also tried and replicated their attack on the NIST code of AES [76] but found no correlation between the frequency of occurrence of cache miss and the encryption time. However, they were able to extract 96-bit key difference, reducing the brute-force search to 32 bits. Unfortunately no detail has been given and no clarification nor advance on their work have been publicly published in order to confirm their claims on AES attacks.

IV.3. Bernstein's Attack

In the setup described by Bernstein [6], a client computer $\mathcal{C}$ is remotely connected to a server $\mathcal{S}$ through the following custom-designed protocol. The client $\mathcal{C}$ generates random plaintexts and sends them to the server $\mathcal{S}$. The latter encrypts them and returns the time needed for the encryption for each of them. As a matter of fact, in this case, the server $\mathcal{S}$ is the one that outputs the execution time measured using the x86's RDTSC instruction (Read Time Stamp Counter, cf. [106]), of the server processor.

In his report, Bernstein presents the technique used in his attack and gives considerations from different angles in order to be out of cache attacks' range. However, no precise indication is given to understand the source of the leakage. The purpose of this Section is to clarify his experiment, to understand why timing differences leak key bits and to lead to the theoretical limit of the attack[1].

[1]This study has been conducted as a part of an internship at Intel Corp. and results have been presented at the rump session of Crypto 2005 [17] and at the Developers' track of the RSA Conference 2006 [36].

IV.3.1. Description of Bernstein's Technique

Bernstein's attack will be mentioned in the following as *first round (timing) attack*. In his report, D. Bernstein gives a description of his results along with the source codes of his attack. His complete key recovery technique is composed of four stages:

$$\begin{tikzcd}
 & \text{Client } \mathcal{C} & \text{Server } \mathcal{S} \\
\text{Learn:} & \mathcal{P}, \mathbf{k} \arrow[r, shift left, "\mathbf{p}_i"] & E_{AES}(\mathbf{p}_i, \mathbf{k}) \arrow[l, shift left, "time(E_{AES}(\mathbf{p}_i,\mathbf{k}))"] \\
\text{Attack:} & \widetilde{\mathcal{P}} \arrow[r, shift left, "\widetilde{\mathbf{p}}_i"] \arrow[d, "\text{partial key recovery}"] & E_{AES}(\widetilde{\mathbf{p}}_i, \widetilde{\mathbf{k}}) \arrow[l, shift left, "time(E_{AES}(\widetilde{\mathbf{p}}_i,\widetilde{\mathbf{k}}))"] \\
\text{Correlation:} & \arrow[d, "\text{full key recovery}"] & \\
\text{Brute force:} & \widetilde{\mathbf{k}} &
\end{tikzcd}$$

Learn phase. Here, the attacker is required to know $\mathbf{k}$. So either he has control over the server or a duplicate[2] of the server (*offline learn phase*), or he has a legitimate communication with the server and therefore knows $\mathbf{k}$ (*online learn phase*). Let $\mathcal{P} = \{\mathbf{p}_0, \ldots, \mathbf{p}_{\ell-1}\}$ be a set of ℓ random plaintexts. The client $\mathcal{C}$ submits each of them, one by one, to the server $\mathcal{S}$. The latter encrypts each plaintext separately and measures each encryption time, *i.e.* $time(E_{AES}(\mathbf{p}_i, \mathbf{k}))$. It then sends the timings back to the client which treats them the following way. A program on $\mathcal{C}$ keeps tracks on the average timing for each byte and each value of that byte in a table $\mathbf{t}[16][256]$ where all the elements are initially initialized at zero. More precisely, for each byte ($0 \leq j < 16$) and all plaintexts ($\mathbf{p}_i$ with $0 \leq i < \ell$), $\mathcal{C}$ successively computes

$$\mathbf{t}[j][p_{j,i}] := \mathbf{t}[j][p_{j,i}] + time(E_{AES}(\mathbf{p}_i, \mathbf{k})).$$

In other words, eventually we have for

$$j \in \{0, \ldots, 15\} \text{ and } x \in \{0, \ldots, 255\}$$

$$\mathbf{t}[j][x] = \sum_{\{i | p_{j,i} = x\}} time(E_{AES}(\mathbf{p}_i, \mathbf{k})).$$

Also, as the plaintexts are randomly chosen, the number of measurements per byte and per value is stored in a table $\mathbf{tnum}[16][256]$, as well as some other statistical information (as standard deviation). After ℓ

[2]The duplication covers hardware, software and workload aspects.

encryptions[3], one can plot the *signature* of each byte — a chart showing the *average* computing time (in processor cycles) for each individual value a byte can take. More formally, the representation shows the average byte processing time for all possible values, subtracted by the overall average encryption time:

$$\mathbf{v}[j][y] = \frac{\mathbf{t}[j][y]}{\mathbf{tnum}[j][y]} - \frac{\sum_j \sum_x \mathbf{t}[j][x]}{\sum_j \sum_x \mathbf{tnum}[j][x]}.$$

See for example Figure 1. The X-axis represents values from 0 to 255 a particular byte can take, while the Y-axis gives the relative timing for each individual value compared to the mean timing. The complete learn profile comprises 16 signatures of this type.

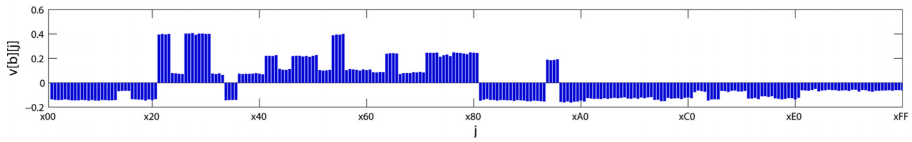


Figure 1: Signature chart for an individual byte.

All together, the learn phase establishes a profile of $\mathcal{S}$ for randomly chosen but known plaintexts and a known key.

Attack phase. Here the attacker communicates with the same server (or a duplicate) but, this time, it is not under his control. Therefore, the secret key $\widetilde{\mathbf{k}}$ used in this case is completely unknown to him. The methodology of the attacker remains the same. $\mathcal{C}$ submits a set of random plaintexts $\widetilde{\mathcal{P}} = \{\widetilde{\mathbf{p}}_0, \ldots, \widetilde{\mathbf{p}}_{\widetilde{\ell}-1}\})$ to $\mathcal{S}$ and treats the encryption time the same way as before. *I.e.* with two tables $\widetilde{\mathbf{t}}[16][256]$ and $\widetilde{\mathbf{tnum}}[16][256]$ he computes

$$\begin{aligned} \widetilde{\mathbf{t}}[j][\widetilde{p}_{j,i}] &:= \widetilde{\mathbf{t}}[j][\widetilde{p}_{j,i}] + time(E_{AES}(\widetilde{\mathbf{p}}_i, \widetilde{\mathbf{k}})) \\ \widetilde{\mathbf{tnum}}[j][\widetilde{p}_{j,i}] &:= \widetilde{\mathbf{tnum}}[j][\widetilde{p}_{j,i}] + 1 \end{aligned}$$

for each byte ($0 \le j < 15$) and each plaintext ($\widetilde{\mathbf{p}}_i$ with $0 \le i \le \widetilde{\ell}$). The definition of $\widetilde{\mathbf{v}}[j][x]$ follows the one of $\mathbf{v}[j][x]$ and attack signatures are generated.

Correlation. In this phase the profiles of the learn and attack phases are combined through a simple correlation (see [52]). Another table $\mathbf{c}[16][256]$ is created and for $0 \le j < 16$ and $0 \le u < 256$ the correlation between $\mathbf{v}$ and $\widetilde{\mathbf{v}}$ is then given by

$$\mathbf{c}[j][u] := \sum_{w=0}^{255} \mathbf{v}[j][w] \cdot \widetilde{\mathbf{v}}[j][u \oplus w].$$

[3]Note ℓ can be arbitrarily chosen but we will see that it drives the resolution of the learn phase.

The elements of **c** are sorted in a decreasing order and the highest correlation results are kept following a deviation threshold[4]. As a consequence, only the highly correlated key values are stored as candidate key byte values (Figure 2).

```
24  0  57 50 51 52 55 53 54 56 4d 48 4a 4b 49 4f 4e 4c 5b 5f 59 5a 5d 58 5e 5c
 8  1  61 63 65 64 66 62 60 67
 8  2  01 03 06 04 07 05 00 02
 8  3  f6 f2 f5 f0 f3 f4 f1 f7
24  4  7c 7f 7d 7e 78 79 7b 7a 65 63 61 67 64 66 60 62 77 75 76 72 73 74 71 70
 8  5  51 56 52 55 54 57 50 53
 8  6  72 74 73 71 77 76 75 70
16  7  67 66 62 61 64 65 63 60 68 6f 6d 6c 6a 69 6e 6b
 8  8  0c 08 0d 0a 09 0b 0f 0e
 8  9  24 25 27 26 20 22 21 23
 8 10  d6 d7 d2 d1 d5 d3 d4 d0
 8 11  9a 98 9b 9c 99 9e 9f 9d
24 12  6f 6c 68 6d 6e 6b 6a 69 77 70 73 74 75 72 76 71 65 63 66 67 60 61 62 64
 8 13  9a 9b 98 9f 99 9d 9e 9c
 8 14  b1 b2 b4 b7 b5 b0 b6 b3
 8 15  13 17 16 11 10 15 14 12
```

Figure 2: Example of the result of the correlation phase. The rows give respectively the number of remaining possible values for the byte, the number of the byte and the exhaustive list of possible values.

Brute force key search. Eventually, using the previous result, the attacker applies a brute force key search. Clearly, the key search space gets narrower as the quality of the learn and attack phases increases. Nevertheless, it is shown later in this Section that there is an asymptotic limit for the profiling phases, *i.e.* any additional encryption time cannot enhance the resolution anymore beyond this theoretical threshold.

IV.3.2. Analysis of Bernstein's Technique

Only little intuition is given in [6] concerning why and how this technique actually works. We examine here some evidences and clarifications about this method and we also explain the partial key recovery from the correlation between the learn and attack profiles.

The description of AES (Section I.2) shows that the plaintext is XORed with the first round key during the first `AddRoundKey` operation. Thus, the input of the system to the encryption is actually either $p_{j,i} \oplus k_j$

[4]The variance is computed for each byte values and the correlation result is kept if its correlation score is close to the maximum correlation value, with respect to its variance. Refer to `correlate.c` source code in [6].

or $\widetilde{p}_{j,f} \oplus \widetilde{k}_j$ respectively for the learn or the attack phase. Actually, Bernstein's method computes the two matrices

$$\mathbf{t}[j][x] = \sum_{\{i|p_{j,i}=x\}} time(E_{AES}(\mathbf{p}_i, \mathbf{k}))$$
$$\widetilde{\mathbf{t}}[j][x] = \sum_{\{i|\widetilde{p}_{j,i}=x\}} time(E_{AES}(\widetilde{\mathbf{p}}_i, \widetilde{\mathbf{k}})),$$

thus averaging out the individual running times of each possible value a single byte $j \in \{0, \ldots, 15\}$ can take. As a result, individual time profiles are arising out of random plaintext encryptions for every byte separately — however, depending on the key $\mathbf{k}$ or $\widetilde{\mathbf{k}}$. Let us introduce the following heuristic.

Heuristic: *the pairs satisfying the equality*

$$p_{j,i} \oplus k_j = \widetilde{p}_{j,f} \oplus \widetilde{k}_j$$

will have a matching timing-profile.

Applying this heuristic naturally leads to a correlation between $\mathbf{t}$ and $\widetilde{\mathbf{t}}$ (or $\mathbf{v}$ and $\widetilde{\mathbf{v}}$). Therefore, possible secret key byte candidates $\widetilde{k}_j$ are easily determined from this "`AddRoundKey` correlation" equality through

$$\widetilde{k}_j = p_{j,i} \oplus k_j \oplus \widetilde{p}_{j,f}.$$

Note that this relation is the same as in the *non-elimination* methodology of Tsunoo *et al.* (refer to Section IV.2).

IV.3.3. Technical artifact behind Bernstein's technique

We have seen so far that small overall timing differences between AES computations featuring

$$p_{j,i} \oplus k_j \quad \text{and} \quad \widetilde{p}_{j,f} \oplus \widetilde{k}_j$$

might lead to a potential key recovery, through the non-elimination relation. However, we still need to precise the source of the leakage and why this attack is possible. The affirmative answer to this question and explanation of why Bernstein's attack is indeed working are closely linked to the behavior and the size of different cache levels.

During compilation of the AES program, the 1kB S-Box tables T_0, T_1, T_2 and T_3 are given addresses in memory, from which their position in the cache will be later derived. During the first run of AES, some cache misses may occur and the corresponding (and surrounding) data and especially the tables are then copied from memory to their respective cache lines. When the encryption is completed, the program executes some other tasks (creation of packets for the client, sending packets; see lines 22 to 24 and 28 of Figure 3) using variables and data, where the cache is also involved[5]. Therefore some of the cache lines previously used by AES might then be evicted and when the AES encryption is re-executed either the accesses to S-Boxes are already resolved in the cache (cache-hit) and the access time is short or they are not (cache-miss) and it takes more time to copy from memory to cache. The total encryption time is therefore clearly a function of the S-Box accesses induced by $p_{j,i} \oplus k_j$ for all $j \in \{0, \ldots, 15\}$, the state of the cache and many other effects (influencing the underlying processor).

```
server.c
...
void handle(char out[40],char in[],int len)
{
    unsigned char workarea[len * 3];
    int i;
    for (i = 0;i < 40;++i) out[i] = 0;
    *(unsigned int *) (out + 32) = timestamp();
    if (len < 16) return;
    for (i = 0;i < 16;++i) out[i] = in[i];
    for (i = 16;i < len;++i) workarea[i] = in[i];
    AES_encrypt(in,workarea,&expanded);
    /* a real server would now check AES-based authenticator, */
    /* process legitimate packets, and generate useful output */
    for (i = 0;i < 16;++i) out[16 + i] = scrambledzero[i];
    *(unsigned int *) (out + 36) = timestamp();
}
...
```

Figure 3: Fragment of `server.c` (refer to [6] for full code) that handles the encryption and its timing.

However most of the cache evictions due to other processes are certainly not completely random, most of them are occurring for constant cache lines. Let us clarify this with an interesting practical example. Figure 4 shows the different cache line accesses when running AES encryptions in a real environment, when many other processes are also executed concurrently to AES. This picture was made by a dedicated software making the different cache-accesses visible per cache-line (X-axis) and at different times (Y-axis), cf. Chapter V for precise measure conditions. The different S-Box accesses are clearly visible in the center. Among interesting features we can observe vertical lines of variable widths (vertical stripes), demonstrating continuous accesses to specific cache lines. This means that the encryption process is interrupted to let other processes being momentarily executed on the processor. Moreover, one sees certain vertical stripes conflicting with the AES S-Box accesses generating the *deterministic* cache-misses and indeed inducing a longer encryption

[5]Moreover the OS and other processes might spend some processor cycles and also use the cache.

times. Moreover, the encryption itself generates collisions within the table lookups: the hit or miss depends on the previous accesses to the SBox Tables.

Now, the crucial observation to make here is the following. The exploitable different AES execution times — when averaged over many but very similar iterations — are due to deterministic and system-dependent cache evictions. By varying over all possible values for a plaintext byte $p_{i,j}$, Bernstein's technique is implicitly searching for those cache evictions by the system. This leads to the small execution time differences for different plaintext inputs.

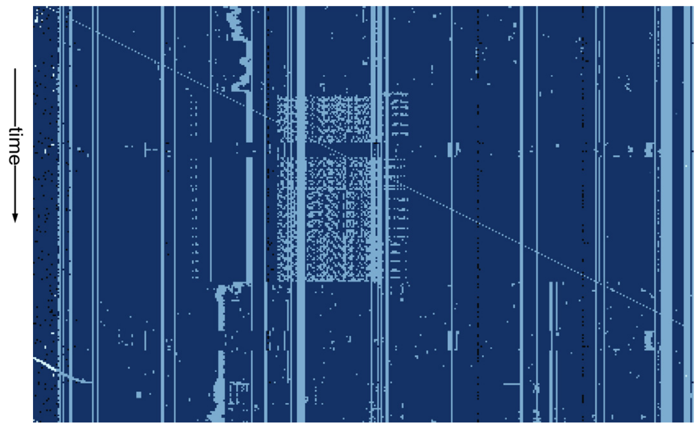


Figure 4: Evolution of the cache *versus* time. Each horizontal line represents the state of the cache lines (represented by a point) at a given time. Different AES encryption are clearly visible in the center. The brighter a point, the longer the time to access its corresponding cache line.

IV.3.4. Theoretical limitations

At this point, it is clear that the constant accesses (vertical stripes) to some cache lines play a major role in the time-driven attack. Specific details have yet to become clearer, such as the role of the associativity and the replacement algorithm; however, Figure 4 brings more clarity on the source of the leakage and allows us to link it with the patterns seen in the profiles (as in Figure 1 and others in the next Section).

From the introduction on caches (Chapter II), we know that a cache miss on one data forces the replacement of a whole L1 cache line by a L2 cache line. Therefore, at the first order, the behavior of all data inside one specific cache line should be the same and their timings should be the same (besides some unavoidable noise in the measure; we will discuss this point later in Section IV.4.2). This observation naturally leads us to the following limitation in resolution of the attack: the number b_{lim}

of indistinguishable bits per byte is linked to the cache line size 2^o by the equation

$$b_{lim} = log_2\left(\frac{2^o}{B_{read}}\right)$$

where B_{read} is the number of byte accessed by the CPU in one cycle: 4 on 32-bit processors (x86) and 8 on 64-bit ones (x86-64). This number explains the *clustering effect* visible in Figure 1: the profile is visibly divided into clusters of 4 values. The validity of this limitation will be discussed in the next Section by confrontations with experimental facts and profiles.

IV.4. Experiments

We give here technical descriptions and observations about experiments we have made on the basis of the source codes given in [6]. Throughout the experiments, we use two 128-bit keys:

- **k**: its value is known by the attacker and here it is arbitrarily fixed with all bytes to `x00`:

```
byte:   0  1   2  3   4  5   6  7   8  9  10 11  12 13  14 15
value: 00 00  00 00  00 00  00 00  00 00  00 00  00 00  00 00
```

- $\widetilde{\mathbf{k}}$: its value is secret and thus unknown of the attacker; we set it in hexadecimal at the following arbitrary value:

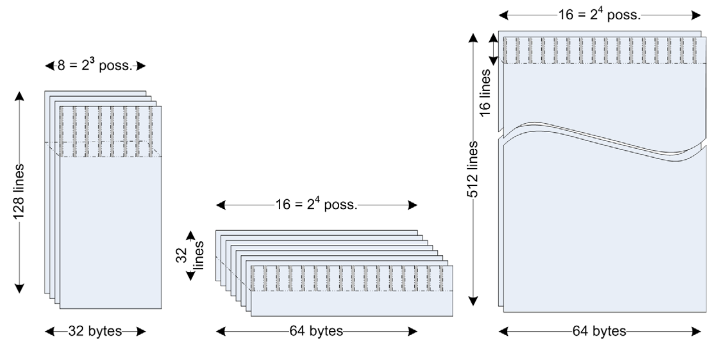
```
byte:   0  1   2  3   4  5   6  7   8  9  10 11  12 13  14 15
value: 57 61  05 f8  7e 51  73 61  08 25  d3 93  68 9a  b3 1b
```

IV.4.1. Processors under test

Here are briefly given the characteristics of the processors used on our experiments (Table 1). The OS of the time measuring program is also mentioned in each case.

Processor	Pentium III E	Pentium 4	Pentium M 735	Opteron 246
Code Name	Coppermine	Prescott	Dothan	Sledgehammer
Technology (μ)	0.18	0.09	0.09	0.13
Core speed (MHz)	500	3000	1700	1993.8
L1 Data Cache				
Size (KB)	16	16	32	64
Associativity (Way)	4	8	8	2
Line size (byte)	32	64	64	64
L1 Inst. Cache				
Size	16KB	2 Kμops	32KB	64KB
Associativity (Way)	4	8	8	2
Line size (byte)	32	NA	64	64
Operating System	SuSE 9.3	XP	XP	Fedora

Table 1: Specifications of the processors under test. XP stands for Windows XP running Cygwin Linux-like environment.



(a) P III (b) P 4 (c) K 8

Figure 5: Details of L1 caches.

The L1 cache characteristics are displayed in Figure 5. The one of Pentium M is omitted as it is the same as Pentium 4 with twice as many lines. The shaded area represents the range of one SBox lookup and the total dashed area, the space required to store a 1KB Table. Note it is only shown on one way but it can be distributed through all of them.

IV.4.2. Preliminary comments on the codes

First, let us have a closer look at the program Bernstein is giving in his report, in order to understand exactly what it achieves.

IV.4.2.1. *Encryption function.* Back to Figure 3 displaying the principal function of `server.c`, observe line 25, *i.e.* when the encryption takes place through an OpenSSL command. Three parameters are required for `AES_encrypt`: `in` the input (plaintext), `workarea` the output (ciphertext)[6] and a reference to the expanded key `&expanded`. Lines 21 and 29 store through the `timestamp()` function (refer to Figure 6) respectively the start and the end values of the `RDTSC` counter, in order to compute the number of cycles in between. However, lines 22 to 24 and 28 (which perform operations on data), are timed, along with the encryption (even if they are not part of the encryption function). They are thus included in the cycle measurement, instead of being exclusively the encryption, even if Bernstein claims: "Of course, I wrote this server to minimize the amount of noise in the timings available to the client" [6]. It is however clear that those data manipulations use the cache and might therefore evict some parts of the previously loaded S-Box Tables.

[6]Note that `workarea` is not used after the encryption, as mentioned earlier.

IV.4.2.2. *Time stamping function.* The RDTSC instruction can lead to erratic results in performance monitoring, which in our case forces the attacker to average over a higher number of encryptions. Intel-compatible processors (including AMD's [2]) integrate a 64-bit counter (Model Specific Register) that increments every clock cycle. It is accessed upon the RDTSC command (equivalent to the machine code on line 9 of Figure 6), which puts the high-order 32-bit into the EDX register and the low-order 32-bit into the EAX register of the processor.

```
study.c
 4  ...
 5  unsigned int timestamp(void)
 6  {
 7      unsigned int bottom;
 8      unsigned int top;
 9      asm volatile(".byte 15;.byte 49" : "=a"(bottom),"=d"(top));
10      return bottom;
11  }
12  ...
```

Figure 6: Fragment of `study.c` (refer to [6] for full code) capturing the RDTSC value.

However, most processors operate under *Out-of-Order* execution, *i.e.* the instructions are re-organized and executed in an order that can differ from the one decided by the programmer [87]. This technique is commonly used in general-purpose processors to optimize the execution flow. The data dependencies are analyzed and non dependent micro-operations can be shuffled with each other. Even if the processor assure the computational results to be accurate (as expected by the programmer), the execution order of the RDTSC might vary between successive runs[7] and therefore the measured cycle difference would also fluctuate.

Note that the CPUID instruction enables a level of *serialization*[8], by flushing the pipeline and therefore forcing all the preceding instructions to be completed before allowing the program to continue (see [44]). However a subtle quirk of this instruction makes its first couple of executions longer than the next ones. At least three consecutive calls are then needed in order to assure a precise measurement[9].

Finally, RDTSC consumes some cycles to be executed. Even if a constant overhead will not change Bernstein's analysis, the overhead can be precisely measured and subtracted for the sake of accuracy.

[7]Under the exact same conditions: cache status, plaintext, key, etc.

[8]The CPUID instruction is usually used to identify the processor running the code.

[9]Remember that instructions too must be cached in order to be rapidly accessed. Complex microarchitectural mechanisms are involved here.

IV.4.2.3. *Measure.* Then examine the fragment of `study.c` of Figure 7. This code is executed on the client's computer and performs the following actions: in an infinite loop (line 42), a random plaintext is generated (line 46) and sent to the server (line 48); from the server's respond, it computes in `timing` the difference of cycles (lines 56-57) and treats it through `tally(timing)` (line 59). Figure 8 gives the code of this last function.

study.c

```
35  ...
36  void studyinput(void)
37  {
38      int j;
39      char packet[2048];
40      char response[40];
41      struct pollfd p;
42      for (;;) {
43          if (size < 16) continue;
44          if (size > sizeof packet) continue;
45          /* a mediocre PRNG is sufficient here */
46          for (j = 0;j < size;++j) packet[j] = random();
47          for (j = 0;j < 16;++j) n[j] = packet[j];
48          send(s,packet,size,0);
49          p.fd = s;
50          p.events = POLLIN;
51          if (poll(&p,1,100) <= 0) continue;
52          while (p.revents & POLLIN) {
53              if (recv(s,response,sizeof response,0) == sizeof response) {
54                  if (!memcmp(packet,response,16)) {
55                      unsigned int timing;
56                      timing = *(unsigned int *) (response + 36);
57                      timing -= *(unsigned int *) (response + 32);
58                      if (timing < 10000) { /* clip tail to reduce noise */
59                          tally(timing);
60                          return;
61                      }
62                  }
63              }
64              if (poll(&p,1,0) <= 0) break;
65          }
66      }
67  }
68  ...
```

Figure 7: Fragment of `study.c` (refer to [6] for full code) responsable for the creation of the packets to be encrypted.

These remarks are given here in order to better apprehend the behavior of the original source code of Bernstein and will lead us to redefine the analysis/attack software in Section IV.4.8.2.

study.c
```
...
void tally(double timing)
{
    int j;
    int b;
    for (j = 0;j < 16;++j) {
        b = 255 & (int) n[j];
        ++packets;
        ttotal += timing;
        t[j][b] += timing;
        tsq[j][b] += timing * timing;
        tnum[j][b] += 1;
    }
}
...
```

Figure 8: Fragment of `study.c` (refer to [6] for full code) ascribing the measured encryption time `timing` according to each byte's value.

IV.4.3. Plaintext length

In his report, Bernstein uses plaintexts of length from 400 to 800 bytes, without justification. One would presume that it may be to increase the encryption time and/or the number of cache misses due to internal collisions. However, back to Figure 8, only the first 16 bytes of the plaintext are treated by the function `tally`. The expected impact is written in lines 24 to 25 of `server.c` (Figure 3): `AES_encrypt` of OpenSSL only encrypts the first 16 bytes of `in` and the corresponding ciphertext is stored in `workarea`. Line 24 copies useless bytes from `in` to `workarea` and therefore only generates memory accesses, *i.e.* cache eviction of SBox Tables.

As our purpose is to understand the leakage, we want to reduce the intentional side-effects. We therefore use in the following short plaintexts, resulting in shorter and more accurate measurements.

IV.4.4. Profiles

As described earlier, the learn and attack phases elaborate two signatures of the hardware under attack, by refining profiles for the first 16 bytes of the first round input (plaintext $\oplus$ key). Figures 9 and 10 display such profiles for a PentiumIII (in increasing order, with *byte 0* in the uppermost position). Recall that the x-axis gives the value of the byte in hexadecimal (from `x00` to `xFF`), while the y-axis gives the number of cycles with respect to the average (over all bytes). In this case, the learn phase is composed of about 2^{30} measurements with an average of 1471 cycles per encryption, whereas the attack phase is obtained by 2^{27} measurements.

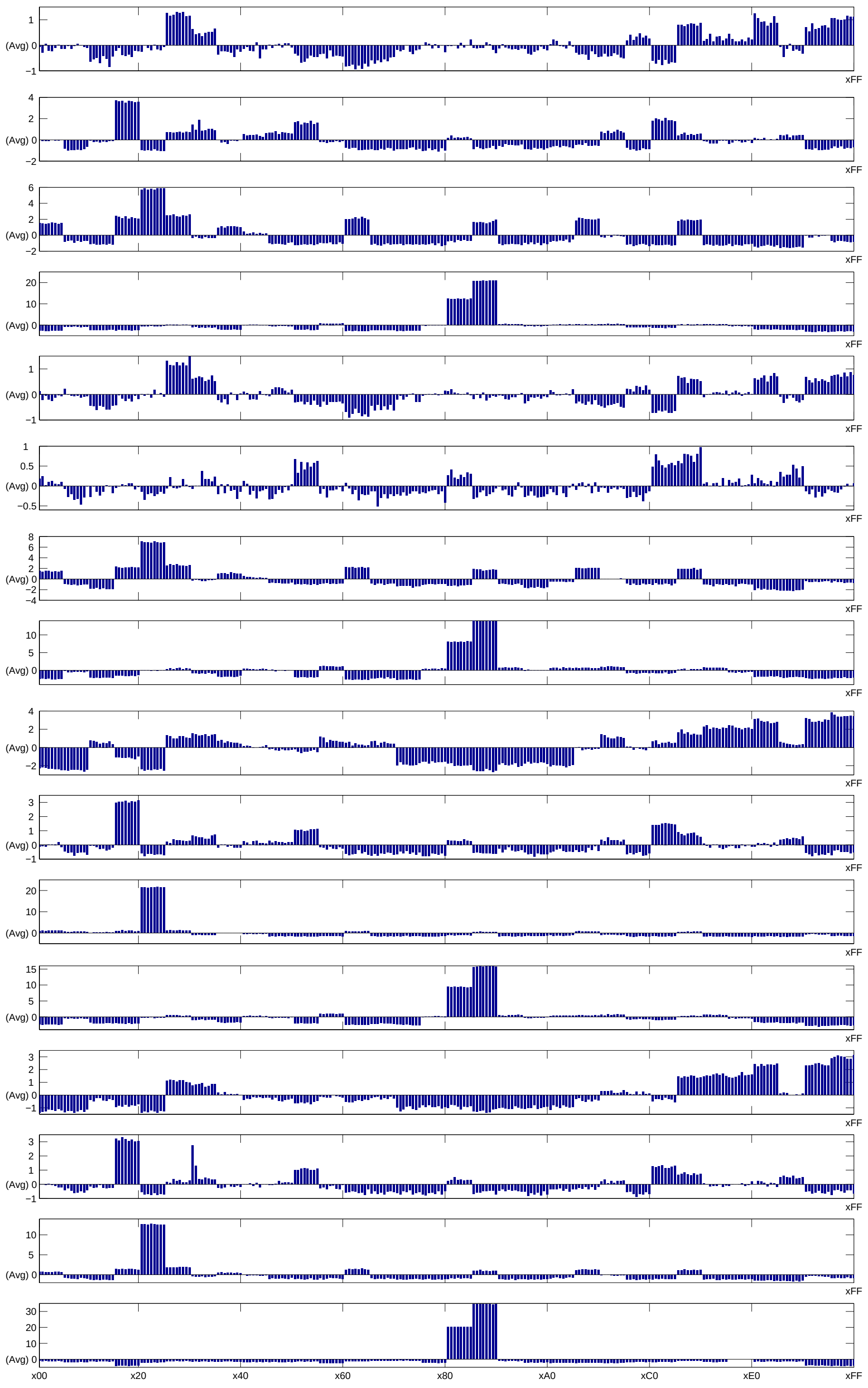


Figure 9: Profiles of a learn phase on the Pentium III.

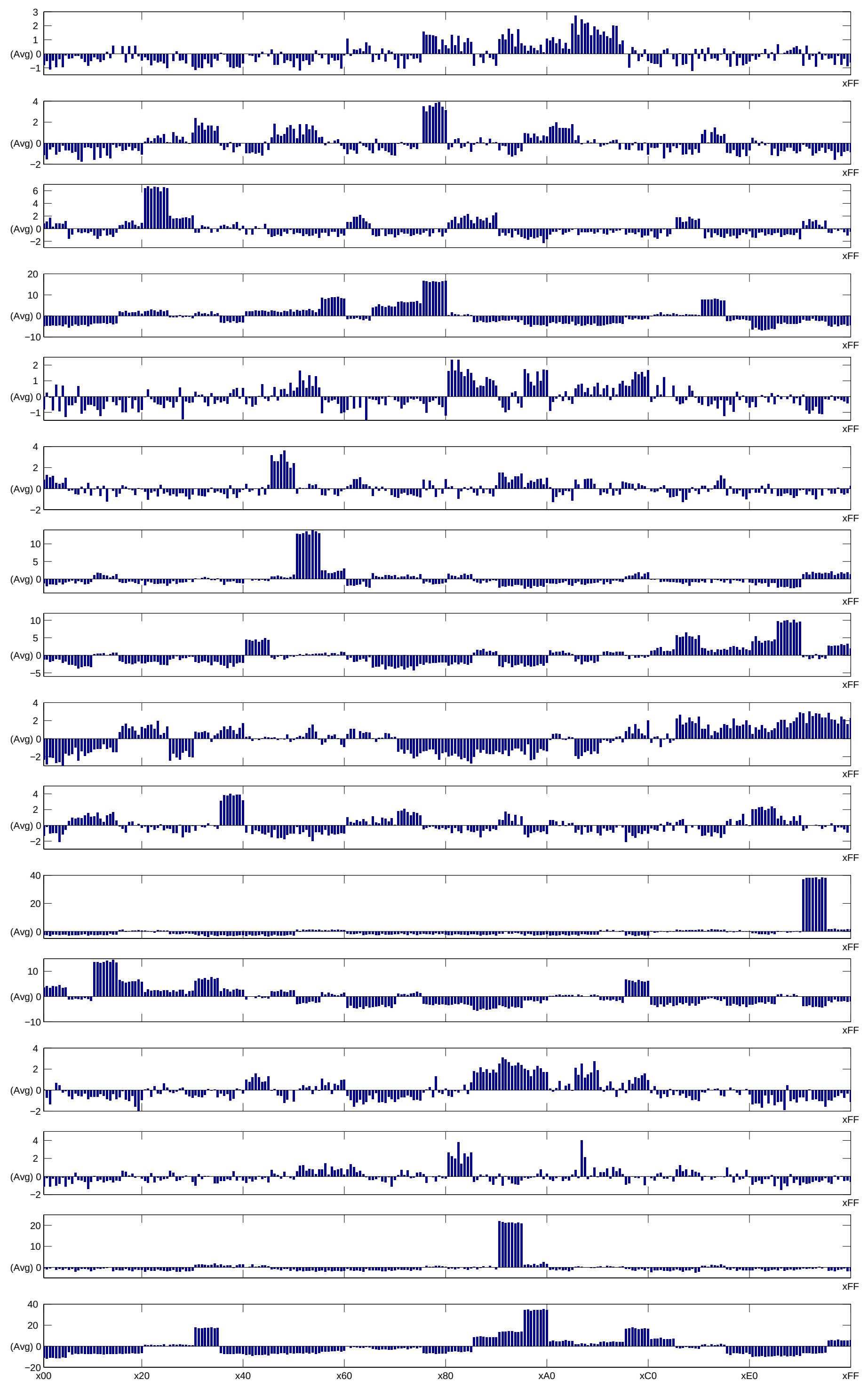


Figure 10: Profiles of an attack phase on the Pentium III.

Each processor we tested has distinct patterns. From now on, we will use the profiles of the PentiumIII to examine general aspects and pinpoint some characteristics; nonetheless other profiles will be shown in the rest of this Chapter.

The result of the correlation phase is the following (Figure 11). The correct value of each byte is underlined and usually stands in the first eight positions. The product of the number of possible values (first column) gives us an estimation of the brute force effort of about 2^{60}. However, let us observe the following facts:

- The correct values for *byte 3* and *byte 15* are not present in the 8 selected possibilities. However, their profiles are similar to the ones of *byte 7* and *byte 11*, for which there are 16 candidate values. The latter have the top value around 15 cycles above average, whereas *byte 3* and *byte 15*'s are around 20. The correct byte values are probably not selected because of the too restrictive threshold in the `correlation` function. Indeed, let us observe that the 4 MSB (*i.e.* the first hexadecimal character) are correct: lowering the threshold would broaden the selection to 16 values and it would include the correct value.
- The estimation of the brute force effort gives a worst case boundary, since it covers the exhaustive list of all possibles keys. A more accurate estimation would take the rank of the correct byte value: the brute force search could start by the most probable key candidates, instead of trying through all candidates of one byte before changing the other bytes' candidate.

```
32  0   50 54 52 55 57 51 53 56 4e 48 4c 4a 4d 4b 4f 49 99 9f 9e 9d 9a 98 9b ..
 8  1   63 61 65 64 62 66 60 67
 8  2   01 00 07 06 03 04 02 05
 8  3   f6 f4 f3 f2 f5 f7 f1 f0 [missing f8]
48  4   7e 7b 79 7f 78 7a 7d 7c 61 65 67 64 60 62 66 63 b0 b1 b7 b6 b2 b5 b4 ..
67  5   80 83 85 84 86 56 82 50 81 51 57 87 55 54 52 53 cc cd 1a ca cb 1b 19 ..
 8  6   71 74 72 77 76 75 73 70
16  7   63 67 64 66 61 62 65 60 69 6a 6f 68 6c 6e 6d 6b
 8  8   0c 0d 09 0f 0e 08 0a 0b
 8  9   27 25 21 22 24 26 23 20
 8 10   d7 d2 d5 d3 d4 d6 d0 d1
16 11   9a 98 9b 99 9c 9d 9e 9f 93 94 97 91 92 96 90 95
24 12   6e 6f 6a 69 68 6c 6d 6b 70 73 71 75 77 76 72 74 63 62 60 65 61 64 67 66
 8 13   9a 9b 9e 98 99 9f 9d 9c
 8 14   b2 b3 b7 b1 b4 b0 b5 b6
 8 15   16 13 12 11 17 15 14 10 [missing 1b]
```

Figure 11: Correlation results for PentiumIII. The correct value is underlined.

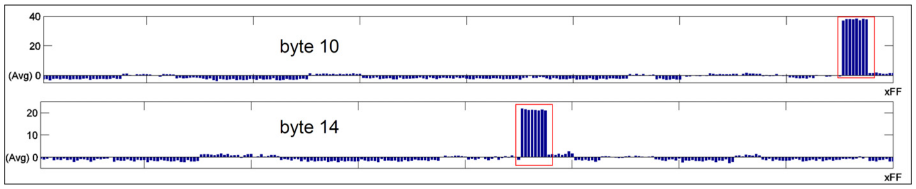


Figure 12: Highlight of similar patterns.

IV.4.5. Similarities between patterns

We already observed the general similarity of some profiles in Figure 9 (the study phase). One can easily see that *byte 0* looks like *byte 4*, *byte 8* and *byte 12*. Usually, the same observation holds for every byte with an equal index i modulo 4, denoted *byte i(mod 4)*. The scale varies inside the *mod 4 classes* but the general shape is conserved, in particular the *clustering effect* (Section IV.3.4).

The similarities are due to the cache architecture[10] and to the fact that all the key bytes have the same value. In the round description of Section I.2, it clearly appears that all *byte i(mod 4)* accesses the same Table $T_{i(mod4)}$ and, for the first round, the same byte within the Table. To demonstrate the patterns' similarities, let us observe Figure 10 and in particular *byte 10* and *byte 14* (Figure 12). The clusters are respectively from `x f0` to `x f7` and from `x 90` to `x 97`. Let us consider only the 5 MSB of each and write them in binary: `b 11110` and `b 10010`, resp. Their difference is computed by an exclusive-OR (XOR, denoted $\oplus$) operation

$$\begin{array}{rr} & \texttt{11110} \\ \oplus & \texttt{10010} \\ \hline & \texttt{01100} \end{array}$$

and corresponds to the difference of the key values for those specific bytes: `x d3` $\oplus$ `x b3` $=$ `b 11010` $\oplus$ `b 10110` $=$ `b 01100`. Similar relations hold for the other *byte i(mod 4)*.

This observation has several consequences:

- Since the attacker knows the value of the key in the learn phase, he can compute the byte-wise key differences and align the profiles for *byte i(mod 4)*. Then he can average those profiles together to increase the resolution of the learn phase, *i.e.* the same correlation results are obtained by performing less encryptions.

[10]In the Pentium III, each way of the cache could hold four 1KB Tables.

- The attacker can deduce key byte relations by solely analyzing the attack profiles. For example, the profiles of Figure 10 have clusters of 8 values. Therefore 3 bits out of 8 are indistinguishable but we can make relations upon the 5 other bits. Hence, if we fix the value of one key *byte i*, the 5 MSB of the other key *byte i(mod 4)* are assigned. Hence, we would have a reduction in the key search from 128 bits to 68 bits[11].

IV.4.6. Seek the highest peak?

The clustering effect that we have already observed tends to gather some values with approximatively the same number of cycles. However, there are small differences of height within the clusters and therefore there is always one highest peak, in the learn as well as in the attack phases. Our experiments show that there is no correlation between the highest peak of both phases: the difference of the byte value of the highest peak does not generally give the correct LSB of the key byte difference.

Recall that the correlation of two profiles A and B outputs the parameters p that produce the maximum values for the sum

$$\sum_j A(j) \times B(p \oplus j).$$

In other words, the function searches the values of p for which the modified $B(p \oplus j)$ profile best matches the A profile. The whole profile is then taken into consideration for this operation. However, the extrema have a greater influence than small values when they are multiplied together. Therefore the byte difference between the highest (or lowest) values leads at the first order to a subset of candidates that have higher probabilities of being correct.

IV.4.7. Easing the correlation

As a straightforward result of the previous paragraph, one can focus on the clusters by performing a preprocessing phase on the profiles before the correlation. The idea is to keep a byte value if its number of cycles above (or below) average is higher than a percentage t of the maximum value. Otherwise, the byte value is set to 0 (average). By this technique, we can generally disclose up to 8 additional bits. However, the choice of t is very sensitive, since it results in reducing the byte candidates. A too restrictive constrain on t would deteriorate the attack's performance.

[11] $68 = 128 - 5$ [bits per byte] $\times (4 - 1)$[*byte(i mod 4)*] $\times 4$[*byte(i)*].

IV.4.8. More realistic setup

Until now, we followed Bernstein's methodology by measuring the time on the remote server. This simplified model was useful here to present the attack and some results but it suffers from inefficiencies. On the one hand, as it is the server that measures the encryption times, the setup can be simplified since there is no need for two computers over a network. A single computer can run a modified program that encrypts, measures and analyzes the data. On the other hand, the analysis over the network should be performed by measuring the encryption times for the client instead of the server. We develop the latter point in the next paragraph and then we analyze a modified version of the code, running on one single computer.

IV.4.8.1. *Measurements over a real network.* In this setup, the client sends packets to the server, which encrypts them and sends them back to the client. Moreover, the client measures the elapsed time between the moment it sends a packet and the moment the encrypted packet arrives. The client is running on Windows XP over the Pentium 4 while the server is the Pentium III running SuSE. We choose this configuration in order to have an increased resolution due to the speed of the Pentium 4. Both firewalls are disabled and the connection is a crossed cable.

Experimental results show, after 2^{30} measurements, noisy profiles with deviation from average of 30 to 50 cycles (to be compared with the 2 to 3 in the previous setup). The correlation does not provide any interesting result. This means that the network delays (back and forth) and their variations decrease the signal to noise ratio. As a result, more measurements would be needed to get useful profiles. But we have to keep in mind that a realistic setup would have security policies requiring to change the secret key, after a defined number of encryptions[12].

IV.4.8.2. *`server_alone.c`.* We come with a modified version of the attack code (see below), combining the main features (encryption and measurements) of `server.c` and `client.c` and leaving out all network aspects[13]. Let us emphasize our maneuver: as the original codes simulate an unrealistic setup where the server gives away the encryption times, we are more interested in quantifying the full potential of a time-driven cache-based attack. This view might vary from the initial purpose of Bernstein but yet unveils practical considerations about realistic vulnerabilities. We will detail them in Section IV.6.

[12]Note however that an attacker could come with an alternative attack taking into account the on-the-fly key change if the precise moment or frequency of change is known.

[13]Therefore, the measurements are also faster to obtain, without affecting their resolution.

server_alone.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <openssl/aes.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <poll.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

double packets;
double ttotal;
double t[16][256];
double tsq[16][256];
long long tnum[16][256];
double u[16][256];
double udev[16][256];
char n[16];

void tally(double timing)
{
...
}

int s;
int size;
void studyinput(void)
{
...
}

int timetoprint(long long inputs)
{
    if (inputs < 10000) return 0;
    if (!(inputs & (inputs - 1))) return 1;
    return 0;
}

unsigned int timestamp(void)
{
...
}

unsigned char key[16];
AES_KEY expanded;
unsigned char zero[16];
unsigned char scrambledzero[16];
void handle(char out[40],char in[],int len)
{
    unsigned char workarea[len * 3];
    int i;
    for (i = 0;i < 40;++i) out[i] = 0;
    if (len < 16) return;
    for (i = 0;i < 16;++i) out[i] = in[i];
    for (i = 16;i < len;++i) workarea[i] = in[i];
    *(unsigned int *) (out + 32) = timestamp();
    AES_encrypt(in,workarea,&expanded);
    *(unsigned int *) (out + 36) = timestamp();
    /* a real server would now check AES-based authenticator, */
    /* process legitimate packets, and generate useful output */
    for (i = 0;i < 16;++i) out[16 + i] = scrambledzero[i];
}

int s;
char in[16];
int r;
char out[40];
int ReadBlock( FILE *infile, unsigned char* buf )
{
        int i;
        for ( i = 0; i < AES_BLOCK_SIZE; i++ )
        {
                int temp;
                if ( fscanf( infile, "%x", &temp ) != 1 || temp > 0xff)
                {
                        fprintf( stderr, "Bad hex value - value above 0xff\n" );
                        return 0;
                }
                buf[i] = temp;
        }
        return 1;
}

main(int argc,char **argv)
{
    char *filename = NULL;
    FILE *infile = NULL;
    int count = 0;
    int j;
    long long inputs = 0;
    char packet[16];
    if (read(0,key,sizeof key) < sizeof key) return 111;
    AES_set_encrypt_key(key,128,&expanded);
    AES_encrypt(zero,scrambledzero,&expanded);
    for (;;) {
        /* a mediocre PRNG is sufficient here */
        for (j = 0;j < 16;++j) packet[j] = random();
        for (j = 0;j < 16;++j) n[j] = packet[j];
        for (j = 0;j < 16;++j) in[j] = packet[j];
        handle(out,in,16);
        unsigned int timing;
        timing = *(unsigned int *) (out + 36);
        timing -= *(unsigned int *) (out + 32);
        if (timing < 10000) { /* clip tail to reduce noise */
            tally(timing);
        }
        ++inputs;
        if (timetoprint(inputs)) studyinput();
    }
}
```

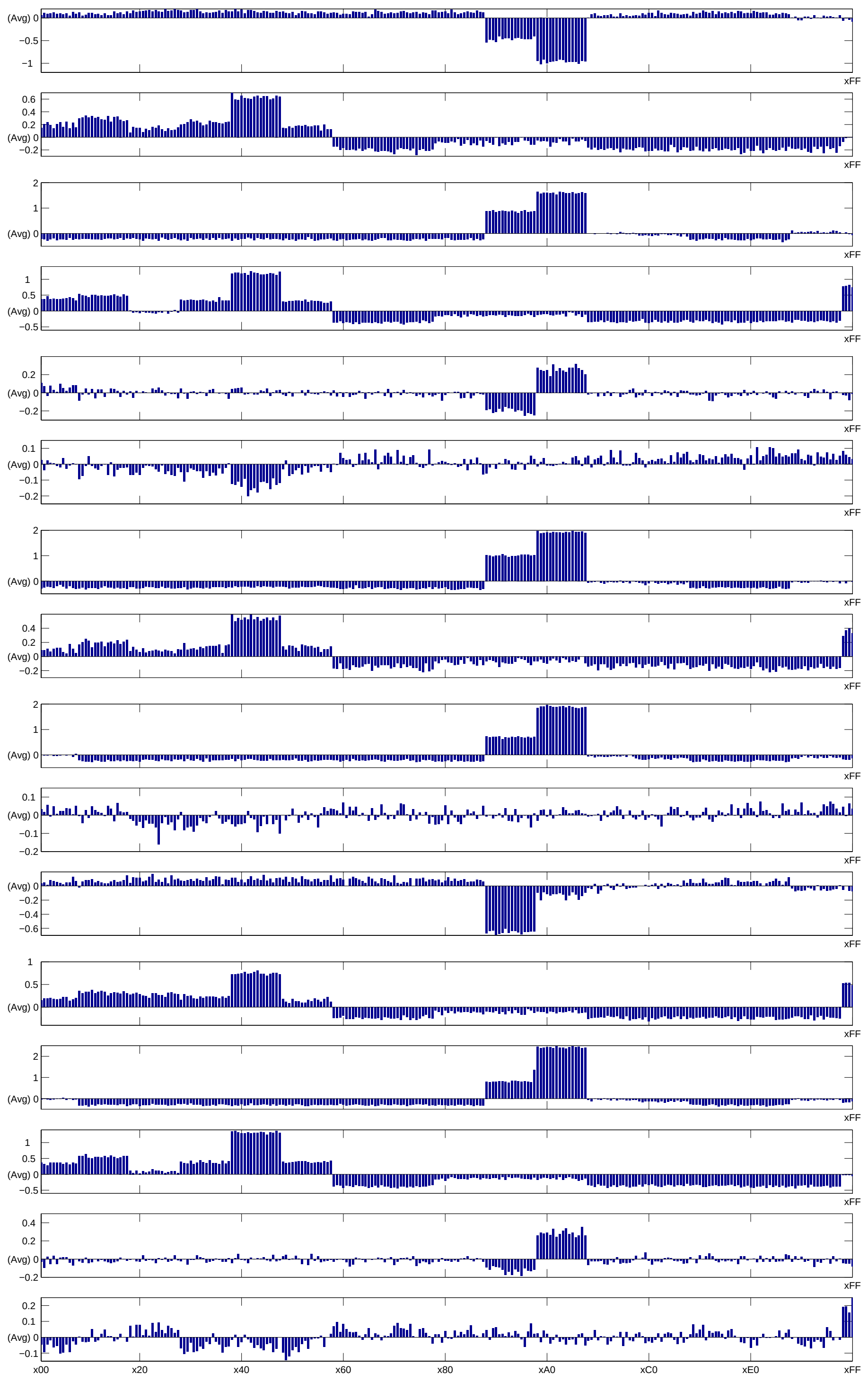


Figure 13: Profiles of a learn phase on the Pentium 4.

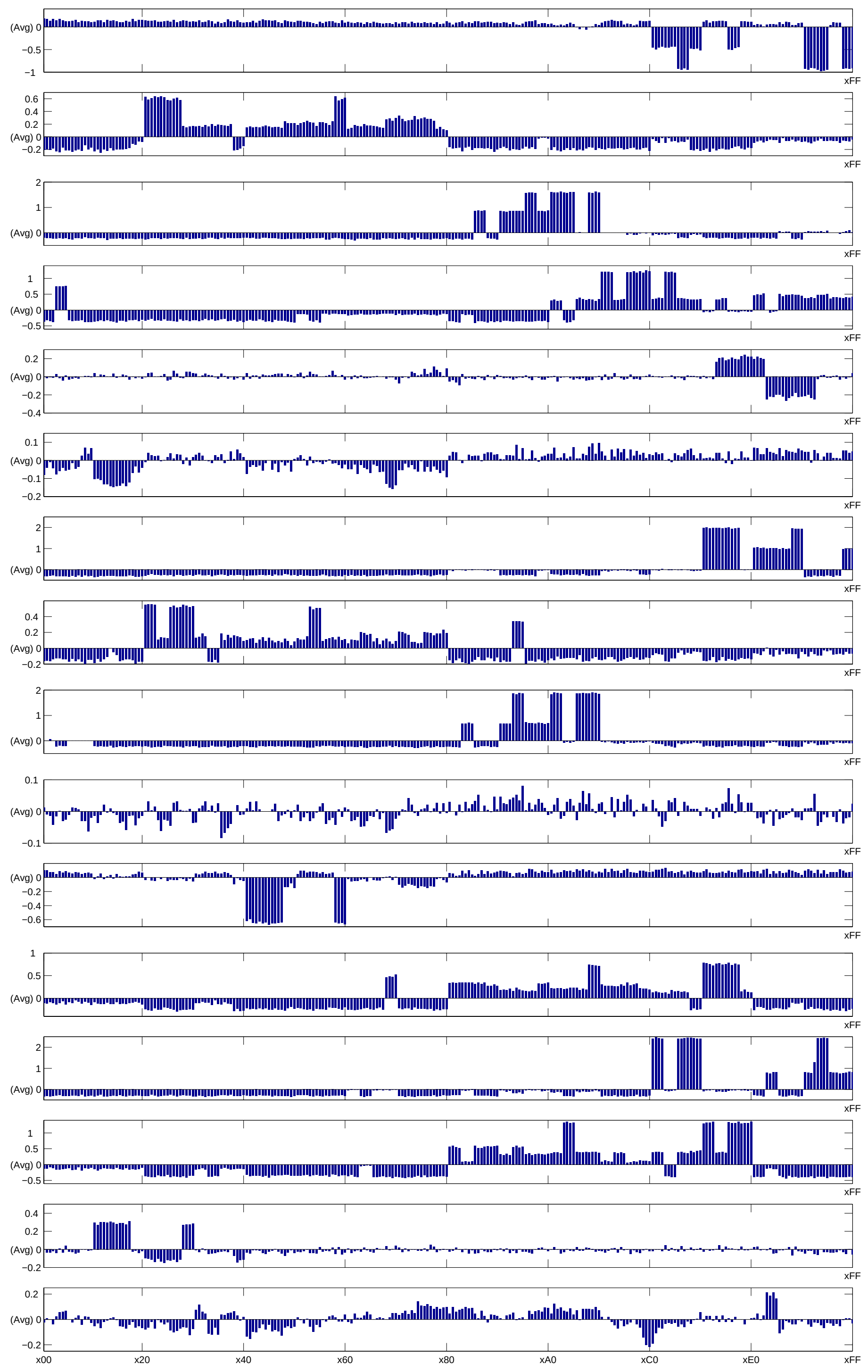


Figure 14: Profiles of an attack phase on the Pentium 4.

We performed experiments on different processors. We give the profiles for the Pentium 4 in Figures 13 and 14 and for the Opteron in Figures 18 and 19.

IV.4.8.3. *Pentium 4.* Let us first consider the learn profiles of the Pentium 4 in Figure 13. They are the results of 2^{30} measurements, with an average encryption time of 695.80 cycles, under Windows XP running Cygwin.

First let us observe that we have two sets of profile: One set $\mathcal{S}_0$ with an even byte numbers, the other $\mathcal{S}_1$ with odd's ones; *i.e.* $\mathcal{S}_i =$ *byte i(mod 2)*.

The profiles of $\mathcal{S}_0$ have two adjacent clusters of 16 values: a first one with small distance to average (in hexadecimal basis, from `x 8c` to `x 9b`) and the second one with greater distance (from `x 9c` to `x ab`). The number of cycles of the clusters can be both over average (as in *byte 2*), both below average (as in *byte 0*) or on both sides (as in *byte 4*). Likewise, profiles of $\mathcal{S}_1$ present a 16-value wide cluster from `x 3c` to `x 4b`. They are generally half a cycle above average. A large part of the contiguous values is below average (from `x 5c` to `x fb`). These two features make those profiles very distinguishable from the others. However *byte 5*, *byte 9* and *byte 15* present noisy profiles, without any specific pattern.

Thus we generally observe clusters of 16 consecutive values. This is in concordance with the 64 bytes per cache line in the Pentium 4: as explained in Section IV.3.4, the placement of the SBox Tables makes similar timing characteristics for all data inside a same cache line. However, let us also notice the 4-value cluster at the end of *byte 3(mod 4)*; one would only clusters of 16 values, given the cache line size. Moreover, the 16-value clusters are not sharing all four first MSB, as one would expect considering the previous experiments on Pentium III. We explain this fact by the dis-alignment of the SBox Tables in the L1 data cache. Indeed, unless explicitly requested[14], the compiler (GCC in this case [116]) aligns data sets with a granularity of 16 bytes. Refer to Figure 15 which represents the cache layout of an SBox Table. In this case, the SBox Table presents a 16-byte offset that dis-aligns the Table. The latter would normally be divided into 16 lines of 64 bytes (16 = 1KB/64 bytes). However, due to the offset, 17 cache lines are covered (the first and the last Table lines occupy only a fraction of a line). We know that the accesses to the data of a particular line usually have the same timing. This is the reason why the offset is then visible on the profiles, as depicted in Figure 15.

The same clusters of 4 values are also visible in the attack profiles (Figure 14) and lead to the disclosure of 6 bits per byte, hence 96 bits of the 128-bit key. The dis-alignment is therefore a crucial fact for the

[14]The option is `-mpreferred-stack-boundary=`*num* to align according to a granularity of 2^{num} bytes.

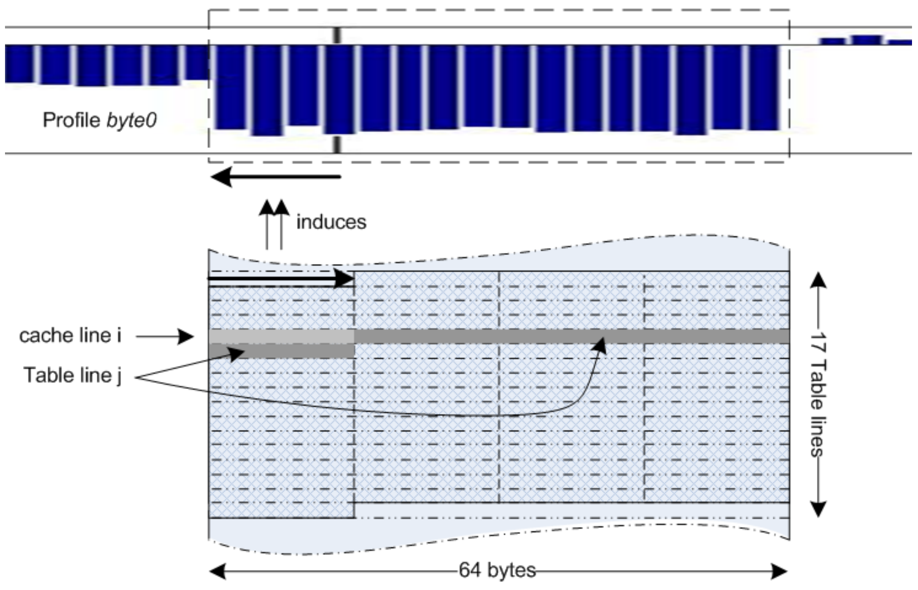


Figure 15: Dis-alignment of a 1KB SBox Table over an L1 data cache of Pentium 4. The initial offset of 16 bytes forces the profiles to be shifted by 4 values to the left.

key disclosure, decreasing the brute force search from a theoretical complexity of 2^{64} (without dis-alignment) to 2^{32} (with dis-alignment). This consideration places the processor in range for a brute force search.

Even better results were obtained on Pentium 4 with profiles presenting a unique peak. A unique byte value is then deduced if both learn and attack profiles of that byte presented the peak. See for example Figure 16 which also shows the evolution of the correlation with the number of measurements.

IV.4.8.4. *Opteron.* The processor of AMD reveals different learn and attack profiles (Figures 18 and 19) with respectively 2^{28} and 2^{27} measurements.

A first particular feature is the repeated patterns visible in *byte 0*, for example. The profile is composed of 4 lower values, 2 higher ones and then 10 close to the average. It is repeated 16 times, uniformly distributed along the profile. The smaller cluster (composed of the two higher values) are at positions `x·a` and `x·b`. The profile of *byte 0* in the attack phase has the same aspect and the two higher values are at positions `x·c` and `x·d` (with · representing any character). The correlation gives then two possible values for the lower half byte: `x·6` and `x·7` (*i.e.* three bits `b····011·` are discovered). Hence, 32 key byte candidates remain,

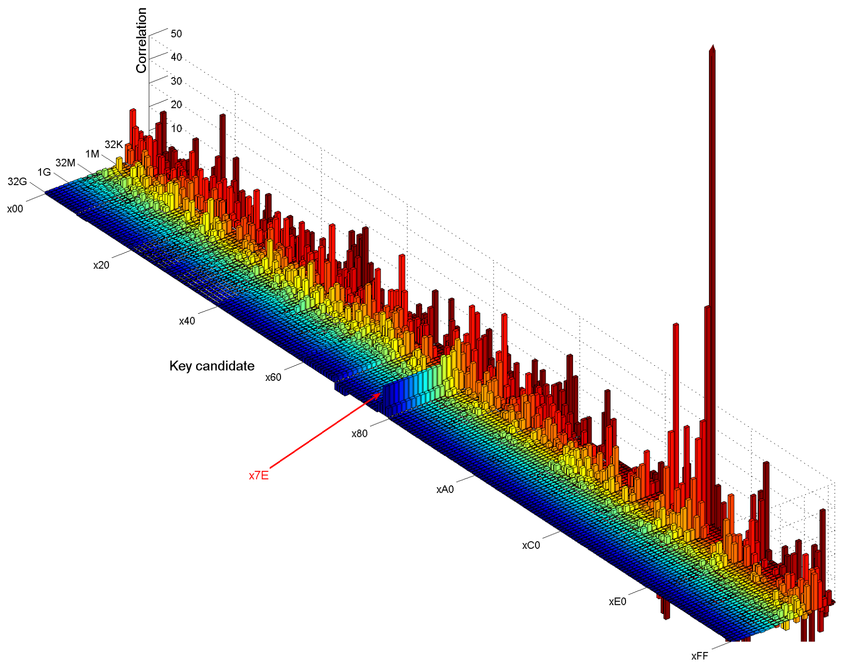


Figure 16: Evolution of the correlation for all values of one byte according to the number of measurements for the attack phase: from 32G to 16K measurements. The number of measurements for the learn phase is kept constant and high.

as shown in Figure 17. Note *byte 3*, *byte 4* and *byte 15* demonstrate the same feature.

The profiles of *byte 1* exhibit a cluster of 8 values starting at positions `x00` and `x60` respectively for the learn and attack phase. This leads to 8 possible key byte values (`x60` to `x67`) including `x61`, the correct one. Other *byte 1(mod 4)* have a similar cluster but the correlation results are different: indeed *byte 5* and *byte 9* reveal only two key byte candidates. This is due to the combined effect of the repeated patterns (like in *byte 0*) and the cluster. The first effect sets 3 bits (*e.g.* `b····000·` for *byte 5*) whereas the second effect selects 8 candidates (by setting the first 5 MSB *e.g.* `b01010···`). The correlation result is the intersection of the results of these two effects (*e.g.* `b0101000·` resulting in `x50` or `x51`). Nevertheless, a closer look to *byte 1* and *byte 13* reveals smaller –but still visible– repeated peaks that enable us to reduce the choice to two candidates[15].

[15]Note the correct candidate is within the first two correlation results, for both *byte 1* and *byte 13*; there is a need for an optimal threshold function.

```
 32  0  77 76 06 07 f6 c7 c6 f7 16 e6 36 17 46 e7 37 47 d6 b6 b7 66 57 97 67 ..
  8  1  61 60 63 67 66 65 62 64
  8  2  05 04 00 03 01 07 02 06
 32  3  d8 c9 78 18 39 88 48 a8 38 79 c8 59 89 a9 98 58 19 d9 f8 09 e8 99 69 ..
 32  4  fe 1e ef 1f ff ee ae 7f 7e af 0e 0f 8e df de 2f 2e 8f bf be ce cf 3e ..
  2  5  51 50
 15  6  72 73 75 74 76 77 70 71 62 63 64 61 66 65 67
249  7  ba 9b aa ca bb fa 8b fb ab da 9a ea eb 2a db cb 4a 5a 8a 4b 5b 6a 7a ..
256  8  78 c8 79 84 19 7d c9 18 88 7c 89 98 85 82 65 74 99 ... [08 at rank 52]
  2  9  25 24
  2 10  d2 d3
 73 11  12 23 52 03 c3 13 d2 02 73 43 53 22 b2 b3 c2 72 63 ... [93 at rank 25]
256 12  58 c0 6e 3e 5d c2 c5 5f 92 46 68 24 de 03 25 b8 93 4a ff 69 5c c7 96 ..
  8 13  9b 9a 98 99 9c 9d 9f 9e
  4 14  b2 b3 a2 a3
 32 15  5a 3a 3b 5b 6a 2b 2a 6b eb ea 1a 1b 7b 0b 7a 4a 4b 0a 8a 9b fb fa 8b ..
```

Figure 17: Correlation results for the Opteron. The correct value is underlined.

All *byte 2(mod 4)*'s learn profiles present a 40-value wide cluster[16], about 2 cycles above average. As in the case of the Pentium 4, the first quarter of the cluster indicates an offset in the layout of the SBox Table. The number of candidates resulting from the correlation varies from 2 to 15.

Another observation is that some bytes (*byte 4*, *byte 7*, *byte 8*, *byte 11* and *byte 12*) display different profiles between learn and attack phases. Their learn profiles are close to the average (± 0.5 cycle) with sometimes 16 repetitive peaks; whereas their attack profiles exhibit two distinct clusters of 8 values, about 4 cycles above average (*byte 4*, *byte 7*, *byte 8* and *byte 12*), with or without jerks[17]. However, we can compute the difference between those bytes (as in Section IV.4.5). The clusters are located

from `x00` to `x07` and from `xf8` to `xff` for *byte 4*;
from `x70` to `x77` and from `x88` to `x8f` for *byte 8*;
from `x10` to `x17` and from `xe8` to `xef` for *byte 12*.

[16]The cluster in *byte 10* is mixed with patterns similar to the ones in *byte 0*, but the cluster is still visible.

[17]The attack profile of *byte 11* on the contrary is noisy but one could see some repetition in the peaks (this is why it has a better correlation result).

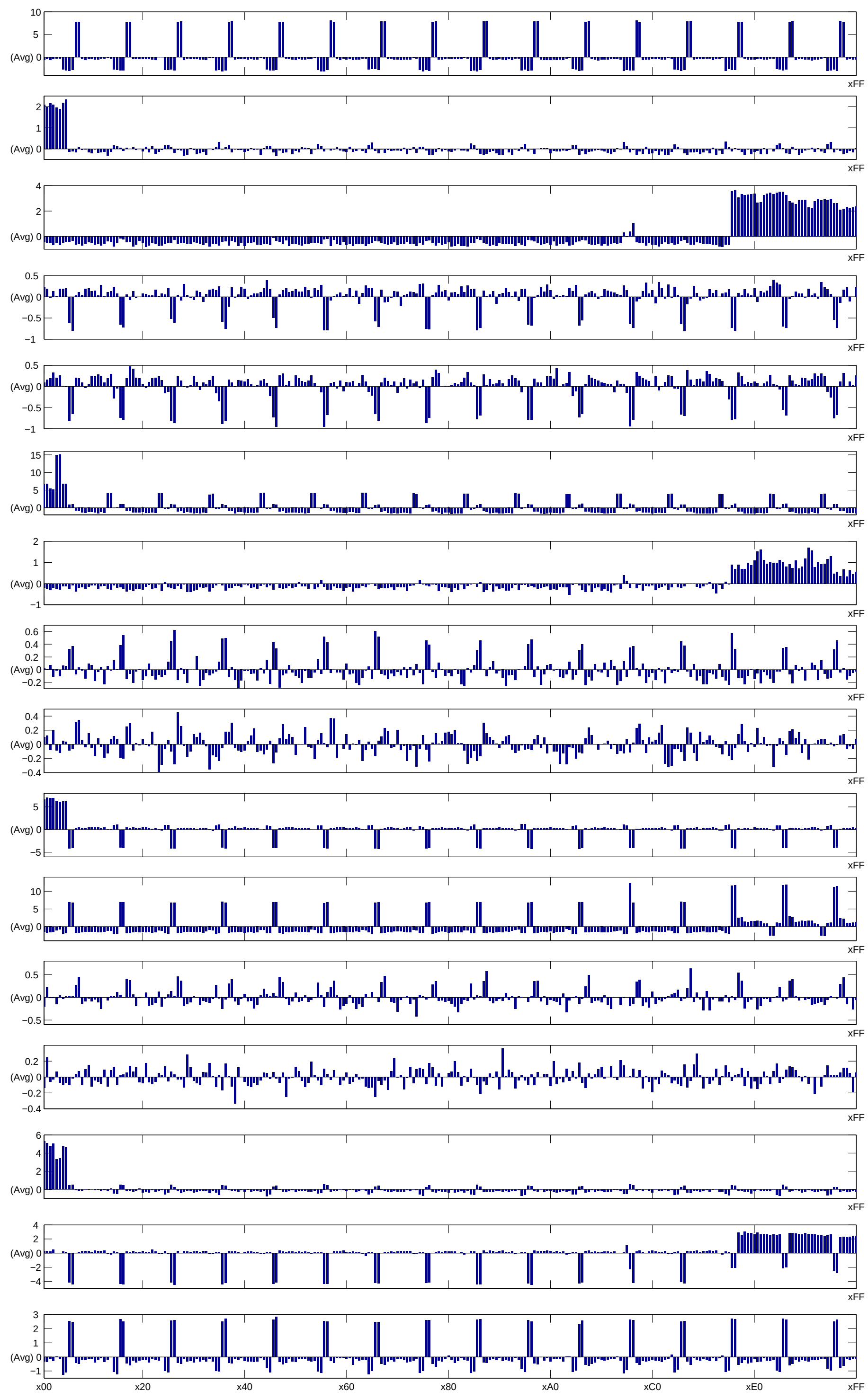


Figure 18: Profiles of a learn phase on the Opteron.

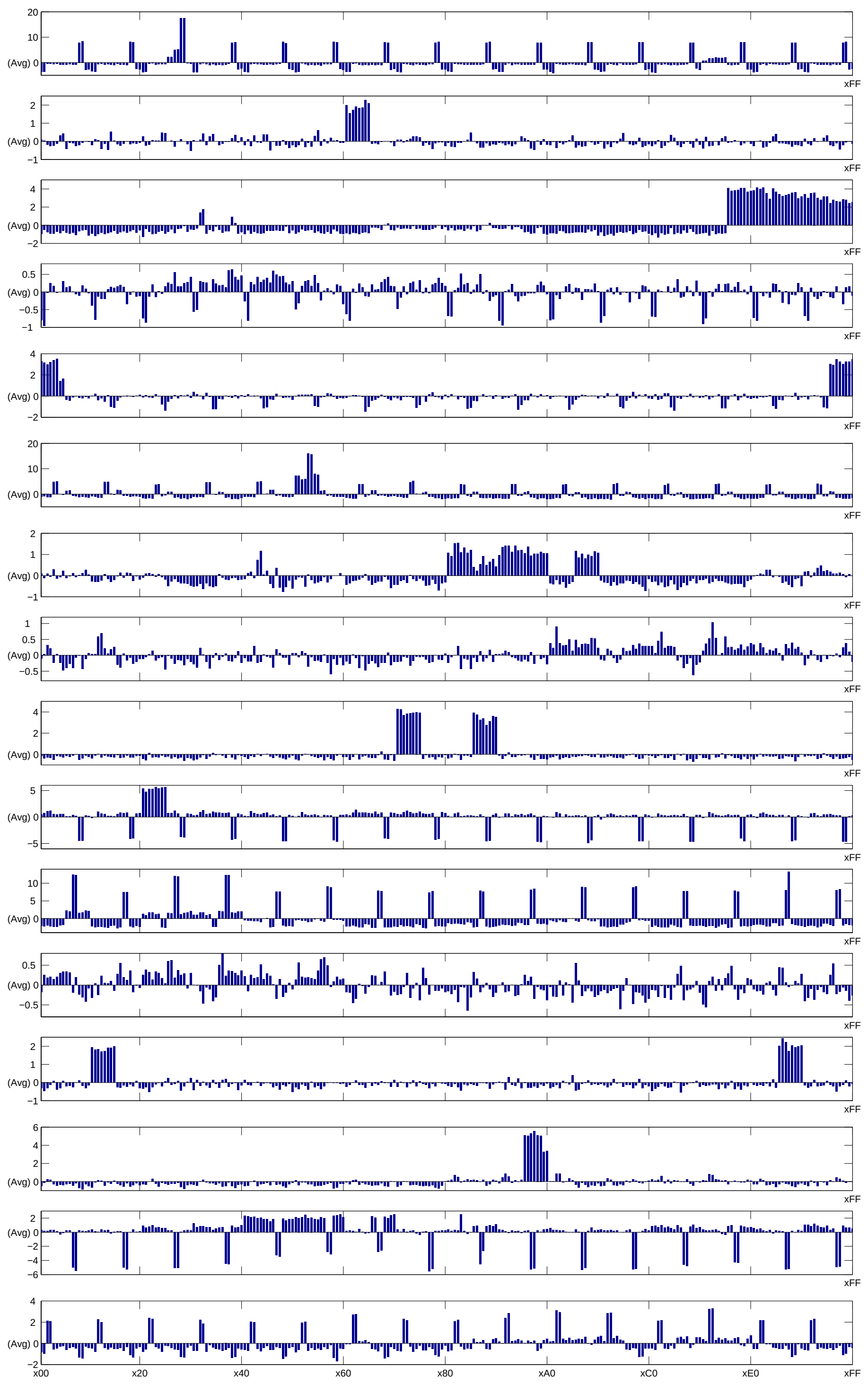


Figure 19: Profiles of an attack phase on the Opteron.

Observe the symmetry of the position of the clusters within a byte, with respect to the center[18]. The difference between the position of the clusters of each byte gives the following key difference (the correct one is underlined):

byte 4 and *byte 8*	lead to	$\texttt{b 10001}\cdots$	and	$\underline{\texttt{b 01110}\cdots}$;
byte 4 and *byte 12*	lead to	$\underline{\texttt{b 00010}\cdots}$	and	$\texttt{b 11101}\cdots$;
byte 8 and *byte 12*	lead to	$\texttt{b 10011}\cdots$	and	$\underline{\texttt{b 01100}\cdots}$.

These considerations enable us to improve the correlation results and reduce the brute force search space to 2^{56}. Even if the cache architecture is very different than the previous ones, it is unclear why the profiles are that noisy under the Opteron. Several points are intriguing and will be the scope of further investigations:

- The repeated peaks leak inner bits of key bytes; the undisclosed bits are an effect of the division in banks of the data cache.
- The symmetry we observed on some attack phase profiles (but not on learn phase) highlights architectural features, that one could exploit.
- As in the previous cases, the dis-alignement can be present. It could occur with a smaller offset, leaking more bits.

IV.4.9. Chosen-plaintext *vs* Random-plaintext attack

Until now, we have considered the case of an attacker who measures encryption times of random plaintexts (*random-plaintext attack*). A more restrictive attack is considered here: we suppose the attacker has the ability to choose the plaintexts sent for encryption (*chosen-plaintext attack*). This assumption might appear unrealistic for the timing attack we started from (practical considerations are discussed in Section IV.6), but our purpose here is to understand the potential advantage of a chosen-plaintext attack. For example, the attacker can then compose the plaintexts in order to infer information: with the assumption of a cache clear of AES Tables, the encryption time is a function of the successive accesses, which are key dependent. Low encryption times infer more internal collisions while longer encryption times demonstrate that more cache lines are accesses.

A first advantage is certainly the possibility of replay, *i.e.* the repetition of the encryption by submitting the same plaintext, with assuring or not similar work conditions (state of the cache, workload, ...).

Chosen-plaintext attacks also enable the attacker to simulate particular points of the encryption. Recall the similarity between the profiles of

[18]For example in *byte 8*, the first cluster position is characterized by the MSB `b 10001` which is the bitwise negation of the position of the second cluster's MSB `b 01110`.

byte i(mod 4) (Section IV.4.5). Let us construct the plaintexts with the following structure

$$R_0\,R_1\,R_2\,R_3\;\; R_0\,R_1\,R_2\,R_3\;\; R_0\,R_1\,R_2\,R_3\;\; R_0\,R_1\,R_2\,R_3$$

which is the concatenation of 4 repeated words and where each word is composed of 4 randomly chosen bytes R_i.

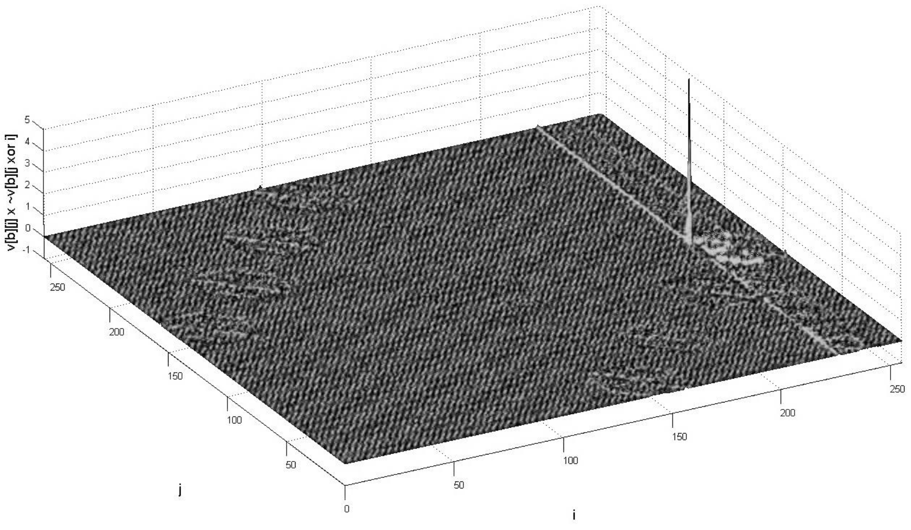


Figure 20: Correlation peak for one byte under a chosen-plaintext attack.

Figure 20 depicts a correlation for a chosen-plaintext attack. The single peak shows that the correlation results in one single key candidate. This single peak can not be explained by our theoretical limitation of Section IV.3.4 and it is therefore the result of others phenomena (*e.g.* internal collisions).

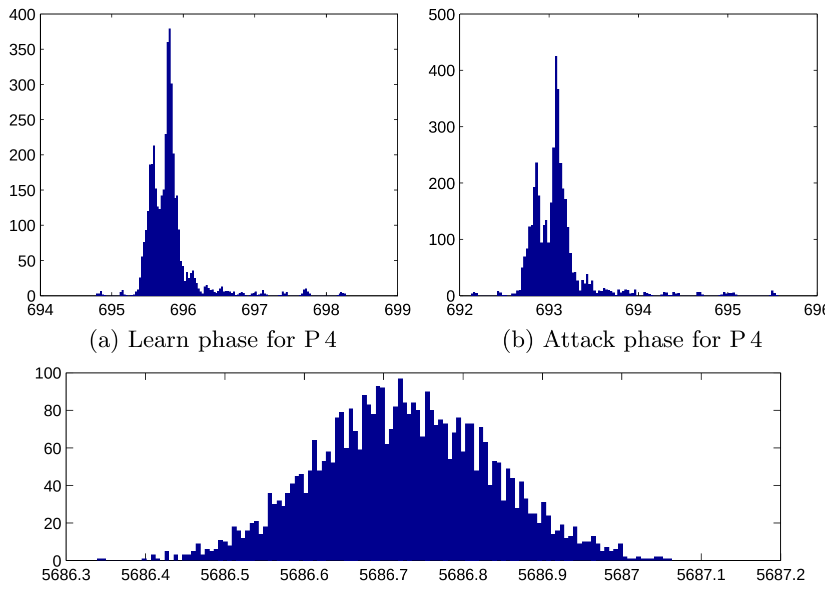
IV.4.10. Distributions

Refer to Figure 21, depicting the distributions of the encryption times for the learn phase (a) and for the attack phase (b) on a Pentium 4 under OpenSSL, as well as a distribution with a mitigated[19] AES code (c) on Pentium III. The distribution acquired on the unprotected code presents a slight shift to the left along with two peaks. The left shift indicates an asymmetry between maximum and minimum values. The double peak exhibits a non-uniform distribution. Although it is not clear how the distribution can help an attacker to deduce information about the key, these profiles obviously enable the attacker to rapidly determine

[19]More details in Chapter VI.

whether or not a further analysis could help: *i.e.* the non-uniformity of the distribution explicitly announces that the timing profiles will disclose useful information. Our experiments were extended to other processors (P III and Opteron) and provide similar distribution.

Conversely, the distribution with a mitigated AES code (Figure 21 (c)) displays a uniform distribution. A thorough analysis of the encryption times did not disclose one single bit. Details on the mitigations are given in Chapter VI.



(a) Learn phase for P 4

(b) Attack phase for P 4

(c) Learn phase for mitigated AES code (on P III)

Figure 21: Distribution of the encryption times for unprotected (a and b) and mitigated (c) AES codes. The distribution is plotted from the averaged data for all bytes.

IV.5. Round 2 Attack

We now describe our *second round attack* as an extension to Bernstein's original *first round attack*. The purpose of extending the attack beyond the first round is twofold. First, from most of the hardware/software configurations we experimented, the key recovery is only effective up to a limited number of bits for each byte. Also increasing the number of encryption measurements is not of any help. But, as we will see, the timing information combined with an additional analysis of the 2nd round can lead to a full key recovery. Second, by exploiting the 2nd round information in combination with the 1st round information, the number of measurements to launch a successful attack can be drastically decreased.

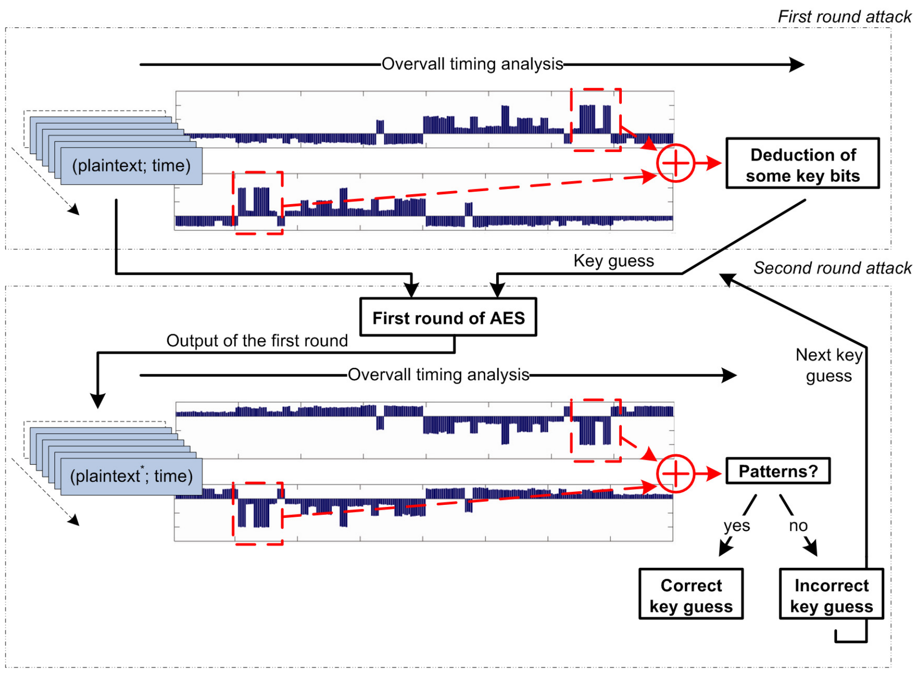


Figure 22: Schematic view of our second round attack.

Let us first describe the idea (Figure 22). Instead of applying the attack only to $\mathbf{p}_i$, we want to extend the attack also to plaintexts $\mathbf{p}_i^{(1)}$ — the output of the first AES round to plaintexts $\mathbf{p}_i$. For the learn phase, the computation of $\mathbf{p}_i^{(1)}$ and $\mathbf{K}^{(1)}$ is trivial. But during the attack phase, $\widetilde{\mathbf{p}}_i^{(1)}$ cannot easily be obtained, because $\widetilde{\mathbf{k}}$ is only partially known — we only have some bits of $\widetilde{\mathbf{k}} = \widetilde{\mathbf{K}}^{(0)}$. But, as shown in Section I.2 each state word $\mathbf{x}_i^{(r)}$ only depends on 4 out of the 16 previous state bytes. For example, the first round computes

$$\mathbf{x}_{0,i}^{(1)} = T_0\left[x_{0,i}^{(0)}\right] \oplus T_1\left[x_{5,i}^{(0)}\right] \oplus T_2\left[x_{10,i}^{(0)}\right] \oplus T_3\left[x_{15,i}^{(0)}\right] \oplus \mathbf{K}_0^{(1)},$$

which transforms the original plaintext $\mathbf{p}_i$ to

$$\begin{aligned}\mathbf{x}_{0,i}^{(1)} &= T_0\left[p_{0,i} \oplus k_0\right] \oplus T_1\left[p_{5,i} \oplus k_5\right] \\ &\quad \oplus T_2\left[p_{10,i} \oplus k_{10}\right] \oplus T_3\left[p_{15,i} \oplus k_{15}\right] \oplus \mathbf{K}_0^{(1)} \\ \mathbf{p}_{0,i}^{(1)} &= \mathbf{x}_{0,i}^{(1)} \oplus \mathbf{K}_0^{(1)}.\end{aligned}$$

Or, in other words, each word of $\mathbf{p}_i^{(1)}$ only depends on four bytes of $\mathbf{p}_i$ and four bytes of $\mathbf{k}$. The same fact also holds for the attack case.

Thus, the second round attack starts by sorting the execution times with respect to each byte of all $\mathbf{p}_i^{(1)}$ from the learn phase and to each byte of all $\widetilde{\mathbf{p}}_f^{(1)}$ from the attack phase:

$$\begin{aligned}\mathbf{t}[j][p_{j,i}^{(1)}] &= \mathbf{t}[j][p_{j,i}^{(1)}] + time(E_{AES}(\mathbf{p}_i, \mathbf{k}))\\ \widetilde{\mathbf{t}}[j][\widetilde{p}_{j,f}^{(1)}] &= \widetilde{\mathbf{t}}[j][\widetilde{p}_{j,f}^{(1)}] + time(E_{AES}(\widetilde{\mathbf{p}}_f, \widetilde{\mathbf{k}})).\end{aligned}$$

In order to do so, the attacker must, for example, compute $\widetilde{\mathbf{p}}_{0,f}^{(1)}$. To make this computationally feasible, he uses the already disclosed key bits from the first round attack and computes all remaining possibilities for the key bytes $\widetilde{k}_0, \widetilde{k}_5, \widetilde{k}_{10}$, and $\widetilde{k}_{15}$. Hereafter, the attacker runs the correlation phase between $\mathbf{t}$ and $\widetilde{\mathbf{t}}$ to recover bits of the round key $\widetilde{\mathbf{K}}^{(1)}$. For the key recovery phase we apply the following simple following heuristic.

> **Heuristic:** a key guess is declared correct, if the correlation produces a meaningful result: *e.g.* if the key space for a byte is reduced to less than 128.

We have chosen this simple heuristic as this "oracle" gives the attacker simultaneously two valuable information:

(1) It rules out most of the wrong key guesses for $\widetilde{\mathbf{k}}$.
(2) It discloses key bits inside the round key $\widetilde{\mathbf{K}}^{(1)}$.

In such experiments the first round attack is not successful in recovering some key bits inside certain bytes and these bytes happen to be the same difficult words during the second round attack, *e.g.* $\widetilde{k}_0$, $\widetilde{k}_5$ and $\widetilde{k}_{10}$ and $\widetilde{k}_{15}$. In this case, the number of possibilities largely increases (up to 2^{32}). To circumvent this issue, the recovered bits of $\widetilde{\mathbf{K}}^{(1)}$ can be used together with the key scheduling algorithm to discard possibilities for $\widetilde{\mathbf{k}}$. This is accomplished by reversing the AES key scheduling in the following way. The first round of the key expansion is given by:

$$\begin{aligned}\widetilde{\mathbf{K}}^{(1)} &= (\widetilde{\mathbf{K}}_0^{(1)}, \widetilde{\mathbf{K}}_1^{(1)}, \widetilde{\mathbf{K}}_2^{(1)}, \widetilde{\mathbf{K}}_3^{(1)})\\ &= (e \oplus \widetilde{\mathbf{k}}_0, e \oplus \widetilde{\mathbf{k}}_0 \oplus \widetilde{\mathbf{k}}_1, e \oplus \widetilde{\mathbf{k}}_0 \oplus \widetilde{\mathbf{k}}_1 \oplus \widetilde{\mathbf{k}}_2, e \oplus \widetilde{\mathbf{k}}_0 \oplus \widetilde{\mathbf{k}}_1 \oplus \widetilde{\mathbf{k}}_2 \oplus \widetilde{\mathbf{k}}_3),\end{aligned}$$

where e only depends on $\widetilde{\mathbf{k}}_3$. Therefore, we get the following equations:

$$\begin{aligned}\widetilde{\mathbf{K}}_0^{(1)} \oplus \widetilde{\mathbf{K}}_1^{(1)} &= \widetilde{\mathbf{k}}_1\\ \widetilde{\mathbf{K}}_1^{(1)} \oplus \widetilde{\mathbf{K}}_2^{(1)} &= \widetilde{\mathbf{k}}_2\\ \widetilde{\mathbf{K}}_2^{(1)} \oplus \widetilde{\mathbf{K}}_3^{(1)} &= \widetilde{\mathbf{k}}_3\\ e \oplus \widetilde{\mathbf{K}}_0^{(1)} &= \widetilde{\mathbf{k}}_0\end{aligned}$$

To recover $\widetilde{\mathbf{k}}_1$, the attacker computes $\widetilde{K}_i^{(1)} \oplus \widetilde{K}_{i+4}^{(1)}$, (with $i = 0, \dots, 3$), for all possibilities obtained from the second round attack and then tries to find the result in $\widetilde{k}_i$ obtained during the first round attack. If there is a match, the corresponding $\widetilde{K}_i^{(1)} \oplus \widetilde{K}_{i+4}^{(1)}$ and $\widetilde{k}_i$ are recorded as possible key bytes. If no match is found, the guess is discarded. To recover $\widetilde{\mathbf{K}}_2^{(1)}$, the same method is applied and the result is then used to update $\widetilde{\mathbf{K}}_1^{(1)}$. Similarly $\widetilde{\mathbf{K}}_3^{(1)}$ can be recovered. Eventually, e is computed using $\widetilde{\mathbf{K}}_3^{(1)}$, and then it is used to recover $\widetilde{\mathbf{k}}_0$.

Experimental Round 2 attack Results
In order to show how useful and viable an analysis of the second AES round really is, we performed the second round attack on different hardware and software configurations which are summarized in Table 2. With our second round attack, all the processors that we tested become under range of a exhaustive search through a narrow search space.

Proc.	# Samples	# bits (1st round)	# bits (2nd round)
Pentium M	2M or more	96	128
	1M	79	125
	400K	56	115
	200K	42	76
	100K or less	(results not stable)	
Pentium 4	8M	101	128
	2M	62	128
	1M or less	(not better than round 1 attack)	
Opteron	2M or more	68	112
	1M	56	80
	less than 1M	(not better than round 1 attack)	

Table 2: Improvements through a second round analysis of AES.

IV.6. Summary

We detail here recent advances on time-driven cache-based attacks. Their starting point is the following. The processor uses the memory hierarchy that has probabilistic timing characteristics, especially due to the cache. The software can be designed to limit the time differences but those could not be completely suppressed without heavy performance drawbacks. Moreover, the capacity of the processor to switch between multiple processes makes it unrealistic for a software to follow the cache status. So, with respect to several factors (work load, memory usage, etc.), a process has different execution times due to the memory hierarchy, more precisely due to the caches. Also the inputs have a special influence since the memory accesses generally depend on them. This feature has a peculiar impact on cryptographic or security-related programs, *i.e.* where sensitive data are utilized as inputs – among others, the secret key. Thus, as the execution time of a program somehow reflects its inputs, the analysis of the execution time leads to the deduction of clues about the key.

Time-driven cache-based attacks naturally belong to the timing side channel family. Ten years ago, the same principle was originally used to guess key structure through the extra-reduction in Montgomery modular multiplication [29, 53, 73], in the framework of smart cards. Although the timing differences induced by the cache mechanism were of common knowledge, several years past until Tsunoo *et al* showed that this time differences could be utilized in a practical attack. They successfully recovered key bits of DES and AES implementation. Then Bernstein detailed his own attack in a report, that was unfortunately written in an obscure way which led the general public consider that to his attack was really done remotely; this should now be clear that it is not the case.

Even if it is clear that Bernstein's work does not represent any danger for remote application, the basis of his work is undeniably useful. Our primary interest was to understand the real nature of the leakage and how much information can an attacker deduce from such an analysis. We then modified the attack scenario to focus on local threads of such cache-based attacks. We believe this scenario to represent a realistic threat. Consider for example the case of security software executed locally on the user processor (banking applications, Digital Right Management, file encryptions and many others): the software is at some extend under the control of the user and he can perform a lot of experiments on it. Of course, measuring the execution time or attacking especially AES is a little part of the threats for processors. Further research directions should investigate chosen-plaintext attacks in the framework of collisions [56, 104].

In this Chapter we enumerated several key points to understand how and why these attacks work. We performed many experiments on different processors and under sundry conditions. We showed the theoretical limitation one can expect in the key recovery, by deepening the cache's properties. We demonstrated the dramatic impact of an offset in the cache. We also developed another attack strategy by suppressing the useless remote features of the original attack. Moreover, we further improved the strategy by extending the attack to the analysis of the second round and performed a real full key recovery. This study was done in order to provide adequate countermeasures presented in Chapter VI.

Further readings

[6] *Daniel J. Bernstein. Cache-timing attacks on AES, 2004. Available online at* `http://cr.yp.to/papers.html\#cachetiming`.

[84] *Dan Page. Theoretical use of cache memory as a cryptanalytic side channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.*

[121] *Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of des implemented on computers with cache. In Walter et al.* [127], *pages 62–76.*

CHAPTER V

Access-driven Attacks

Abstract: An access-driven attack is a class of cache-based side channel analysis. Like the time-driven attack, the cache's timings are under inspection as a source of information leakage. Access-driven attacks scrutinize the cache behavior with a finer granularity, rather than evaluating the overall execution time. Access-driven attacks leverage the ability to detect whether a cache line has been evicted, or not, as the primary mechanism for mounting an attack. In this Chapter we show that the vast majority of processors suffer from this cache-based vulnerability. Our best results are indeed performed on a processor without the multi-threading capabilities — in contrast to previous works in this area that had suggested that multi-threading actually improved, or even made possible, this class of attack.
Despite some technical difficulties required to mount such attacks, our work shows that access-driven cache-based attacks are becoming easier to understand and analyze. Also, when such attacks are mounted against systems performing AES, only a very limited number of encryptions are required to recover the whole key with a high probability of success.

V.1. Page's theoretical attack

In [84] D. Page described the first cache-based attack on DES. Even though it was theoretical, because he started from the hypothesis that an attacker could observe the cache behavior through power consumption or electro-magnetic emanations, his attack drew attention toward the potential of *cache behavior analysis.* He recovered bits by correlation of information on the first two rounds, weakening a 56-bit DES key to an equivalent security of a 32-bit one. His observation called for careful hardware designs and software implementations on embedded systems: the SBox is considered to be the element of a cipher bringing resistance against cryptanalysis [25] but it ruins the cipher's strength in codes that are unaware of cache-based threats. Page gave countermeasures against cache attacks in the report [84] and in [85, 86].

V.2. Percival's Attack on RSA

C. Percival presented in [88] a trace-driven attack on RSA [98], taking advantage of a particular feature of Pentium 4. Although a team at Intel Corp. later extended his work on any processor, we will first give a short overview of NetBurst micro-architecture in order to present Percival's idea.

V.2.1. Simultaneous Multi Threading

NetBurst was deployed in Pentium 4 processors and, in our current concern, presents the feature of multiple execution threads (registered as *Hyper-Threading* (HT) by Intel). HT enables instructions to be reordered across two threads on a single processor and therefore to alternate between two representations of the CPU's registers. As a result, the OS can operate as if there are two logical processors under which two threads can simultaneously be executed. The high pipeline level generates an important delay if one thread stalls (*e.g.* waiting for a data after a cache-miss); with HT, the CPU switches to the second thread and can then keep the pipeline full. According to Intel, the advantage with HT is up to 30% in resource utilization.

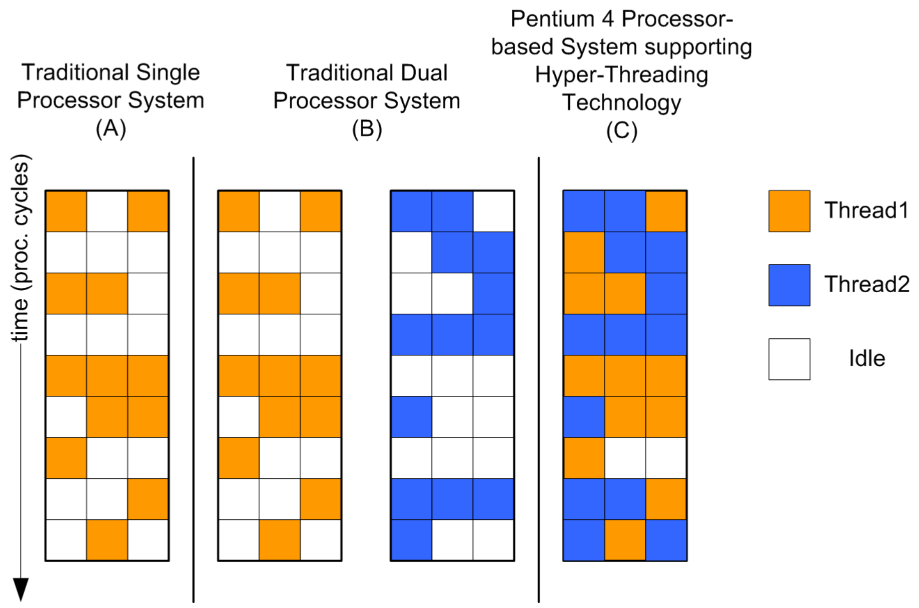

Traditional Single Processor System (A)
Traditional Dual Processor System (B)
Pentium 4 Processor-based System supporting Hyper-Threading Technology (C)
time (proc. cycles)
Thread1
Thread2
Idle

Figure 1: Example of the resources' utilization in various processors. Source Intel Corp.

V.2.2. A Simple Timing Covert Channel

Like the execution units, the memory hierarchy is shared between the threads and this can be viewed as a timing covert channel. The threads cannot communicate by loading and reading data into the L1 cache because the data are private and each cache line is tagged by its owner thread. However, a thread can evict the other thread's data out of L1. An extra latency due to loading from L2 is measured when reaccessing the evicted data. With this simple channel, a high clearance level can then transmit 400KB per second to a low clearance level thread, on a 2.8GHz Pentium 4.

V.2.3. Side Channel

Likewise, a side-channel can take place. The high clearance level thread is now a 1024-bit RSA encryption using OpenSSL [79], while the low clearance level (*Spy*) thread continuously reloads a table of the size of L1 and measures the access time of each cache line. *Spy* outputs a visualization of the L1 activity, hence traces of the RSA signature.

OpenSSL uses the Chinese Remainder Theorem [52, 93] to decompose the 1024-bit modular exponentiation over $\mathbb{Z}_{pq}$ into two 512-bit modular exponentiations over $\mathbb{Z}_p$ and $\mathbb{Z}_q$. Also, it implements a sliding window method, performing a modular exponentiation $x := a^d \bmod p$ into a series of squaring operation $x := x^2 \bmod p$ and multiplications $x := x \times a^{2k+1} \bmod p$. OpenSSL precomputes the set of possible multipliers $\{a, a^3, a^5, \ldots, a^{31}\} \bmod p$ for efficiency matters and stores the values in memory.

A thorough inspection of the cache traces reveals two kinds of information. First, observing the evolution in time of cache line accesses, one observes repeating patterns which can easily be deduced as squaring operations. Indeed, sliding window method transforms the exponentiation in a series of frequent squaring with some infrequent multiplications.

Then, observing the multiplication traces provides information on which precomputed multiplier has been used. Since a multiplier holds into 512 bits, it is represented as $512/8 = 64$ bytes, which is the size of a cache line in Pentium 4. Therefore, the location of the accessed cache line in the multiplication traces indicates which precomputed value has been loaded.

Considering the noise of the traces, C. Percival reports that his attack provides 310 bits average out of 512 per exponent. This information is sufficient to factor the RSA modulus without extensive computation.

Note that this attack is labeled as an access-driven attack, even if the traces are obtained through latency measurements of cache accesses. Time-driven attacks are less sophisticated in their techniques: in those ones, an attacker profiles the behavior of the system without needing much internal details and the software effort is small. In access-driven attacks however, a good knowledge of the hardware is necessary as well as heavy software engineering skills. These considerations make an access-driven attack generally much more complex than a time-driven attack, but also increase its efficiency and impact.

A simple countermeasure to Percival's attack is to organize column-wise the precomputed values. This way, each access to a precomputed value reloads all the cache lines (that holds the precomputed values) and therefore one cannot distinguish one precomputed value from another [36].

V.3. Attacks of Osvik *et al.*

Independently from D. Bernstein, C. Percival and ourself, another team was concurrently studying cache behavior analysis. D. Osvik, A. Shamir and E. Tromer presented several versions of their work at rump sessions or informal presentations as well as technical or proceeding reports [80, 81, 82]. We briefly give the essence of their contributions.

V.3.1. Synchronous known-data attacks

The first type of attacks requires that the plaintext or the ciphertext is known by the attacker and also that this attacker is able to operate synchronously with the encryption, on the same processor.

Osvik *et al.* first introduce a complex analysis of a theoretical attack based on *first round* information. They use an ideal predicate $\mathcal{Q}_k$ over random but known plaintexts to extract cache information. Their model discloses some bits of the key k but only the MSB of each byte may be found. As explained earlier in the context of Bernstein's attack (Section IV.3), contiguous data in a cache line are brought and evicted together and there should not theoretically be any access time difference between each other. Still in a theoretical concept, they take some noise into account by using measurement score distributions $\mathcal{M}_k$ which approximates the ideal predicate $\mathcal{Q}_k$.

A second round attack is then proposed, again with $\mathcal{Q}_k$ and $\mathcal{M}_k$ predicates. A full AES key recovering requires approximately 8220 samples and a complexity of 2^{29} simple tests with $\mathcal{Q}_k$. Moreover they claim that 7 times this number of samples is enough in the $\mathcal{M}_k$. Unfortunately they do not give any detail nor specification of their attack system; hence lacking evidence of tangible results on real systems.

They then give their methods for extracting cache information. They are called Evict+Time and Prime+Probe; they are all composed of three distinct stages. In Evict+Time, the attacker (1) triggers an encryption of a plaintext $\boldsymbol{p}$, (2) he then evicts the SBoxes by accessing some memory addresses and eventually (3) measures the execution time of a new encryption of the same plaintext $\boldsymbol{p}$. In this case, the cache information is similar to Bernstein's measurements, but less general since the same plaintext must be encrypted here (note that they mention this fact and simply adapt to it). They performed a full AES key retrieval with OpenSSL library calls, however once again without further detail. As aforementioned, we obtained similar results and the full details are given in Section IV.3.

The Prime+Probe method presents the following steps: (1) the attacker primes the cache by loading a table as large as the cache size, (2) run an encryption of a random but known plaintext $\boldsymbol{p}$ and (3) re-accesses the table and measures the time for each cache line. This method enables the attacker to construct a trace of the result on the cache of encryption of $\boldsymbol{p}$. Indeed, the encryption of $\boldsymbol{p}$ will evict some cache lines of the attacker's table and he can detect the eviction by the time required to reaccess each data. The attacker can then infer information about the key by studying the SBox accesses of $\boldsymbol{p}$. Note that the cache trace depicts the total effect over all rounds and the resulting picture will be less detailed than Figure 4 of Chapter IV. We will show later how we captured such an image and how its analysis has lead us to a more devastating attack.

V.3.2. Asynchronous attacks

In this case, the attacker does not know the plaintext to be encrypted and the encryption is independent of him (launched by a non-controlled process). Nevertheless, to mount a successful attack, Osvik *et al.* need to add constrains on the hardware under attack, *e.g.* they require a simultaneous multithreaded processor (they discuss possible extensions to SMP processors). Using, as C. Percival did, the shared cache feature of such processor, they obtain statistical profiles of the frequency of cache set accesses. Once they have the knowledge of the frequency distribution of the plaintexts (but not the particular value of each of them), they correlate the frequency distribution with the statistical profiles to extract key bit information.

V.3.3. Experimental results

Under synchronous known-data attacks, Prime+Probe method was reported to full 128-bit AES key recovering after 300 encryptions on Athlon 64 and after 16 000 encryptions for Pentium 4E. They also tested their attack under a Linux `dm-crypt` (*i.e.* an on-the-fly data encryption software) and infer the 128-bit AES key after 800 encryptions (65ms) and 3 second of analysis. However, note that they chose Electronic Code Book

(ECB) as the mode of operation. Using another mode of operation can make the attack much harder. Practical cases generally avoid ECB.

Evict+Time enabled them to find the secret key after 500 000 samples using OpenSSL on an Athlon 64.

Finally a synchronous attack with OpenSSL on Pentium 4 HT provides on average 45.66 bits of the key in 1 minute. No result is given concerning the time or effort to experimentally recover the whole secret key.

V.4. Attacks on single-threaded processors

Access-driven side channels consider that two (or more) processes are executed quasi-parallel on the processor. One process (called here the crypto process) is performing a cryptographic function (*i.e.* AES in this case) involving a secret key. As aforementioned, precomputed values are involved in the execution of the crypto process and their accesses are done through the memory hierarchy. On each data request, the cache checks whether it holds the data, or not. If it does, a cache-hit occurs and the data is immediately transmitted to the processor. Otherwise, a cache-miss occurs and the data must be fetched from a higher memory level, with a longer access time.

A second process, called a spy process, *spies* on the cache accesses of the crypto process. It continuously loads a table S of the size of the cache. From time to time, the crypto process is executed and it inevitably evicts some parts of S by accessing particular data. Therefore, the next time that the spy process is executed, the access time of each part of S (*i.e.* the time necessary to reload a given part of S) indicates which part has been evicted by the crypto process during the last execution of the crypto process.

Thus, the cache is leaking information about the crypto process's memory accesses. Since the software implementation is known, an attacker can infer partial knowledge of the secret key. It is however worth underlining the fact that the spy process cannot diretly access the data of the crypto process; it only observes the cache activity generated by the crypto process and deduces (partial) information from this activity.

V.5. Multi-threaded *vs* single-threaded

Recall that previously described attacks [81, 80, 82, 88] take advantage of the multi-threading capacity of certain processors. It allows them to have two processes running *quasi* parallel on the same processor, as if there were two logical processors [107, 109]. In this manner some logical elements are shared, while the quasi parallelism enables one process to *spy* on the other through the use of the shared logic elements. The cache architecture is one such example of a shared element. Although hardware-assisted multi-threading seems to be mandatory at first sight, we show in the rest of this section that it is not.

Although single-threaded processors run threads/processes serially, the OS manages to execute several programs also in a quasi parallel way, only at a coarser resolution, cf. [109]. The OS basically decomposes an application into a series of short threads that are ordered with other application threads. The processor's resources are thus temporally shared according to the OS's ascribed prioritization.

In order to transfer the (hardware-assisted) multi-threaded processor attacks from [80, 81, 82, 88] to single-threaded processors, one has to leave the comfort of hardware-assistance and exploit subtle OS particularities — which may vary from OS to OS. While this seems quite possible for attacks such as [88], the very fast execution time of AES seems to require the aforementioned hardware-assistance in order to efficiently switch between the spy and the crypto process. Indeed, the objective is to ensure that the crypto thread runs only for a small amount of time between any two runs of the spy thread, or in other words we are able to implement the following strategy:

spy: Continuously watches the cache usage of the parallel crypto thread.

crypto: Runs only for a small amount of time between any two runs of spy.

Interestingly enough, the basic idea is already pointed out in one of the fundamental papers on cache-based side channel attacks, cf. Hu [42], and can adapted to today's OS to stretch the AES execution time over several OS quantums, cf. [42, 109]. According to cf. [42], the so called *preemptive scheduling* property, cf. [109], "...allows a process to control when it yields the CPU to another process without waiting until the end of the quantum." Therefore, using the Linux command `sleep` instead of the VAX security kernel command `WAIT` and the following repetitive spy process paradigm we are able to achieve an implementation of the above attack strategy:

- Watch the cache usage.
- Spend most of the OS quantum.
- Yield the CPU to another process via an appropriate `sleep` near to the quantum end.

This paradigm uses the fact that the OS will reschedule the (very short) remaining quantum part to the crypto thread which will be able to execute a few instructions, after which the OS will quickly reschedule to the spy thread, allowing him to spy on the recently used memory accesses. As the above paradigm and all its subtle implementation details heavily depend on the underlying OS, CPU type and frequency, etc. we will not deepen further this technical details here.

Figure 2 shows the successful implementation of the above strategy and was actually created by observing an unmodified AES implementation through the cache accesses. In this Figure, there are 80 columns. Each column represents a single cache line. The 80 columns are divided into five tables of 16 cache lines, each table representing an SBox (starting with T_0 at the left and continuing to T_4 at the right). Each row in this figure indicates a different measuring time (the uppermost being the first measurement). Each point in a row displays the activity of the particular cache line that it represents. The brighter the point, the longer the time it takes to access an element in the cache line. It is important to understand that we get no information about the order of the accesses within one measurement. By using this kind of picture, an attacker can follow the activity of one or more specific cache lines.

Figure 2 depicts 4 successive AES encryptions. In this particular example, each encryption is repeated 5 times. The time resolution enables us to perform a few measurements per encryption. However, we do not have any distinction between the AES rounds. We only know that they are interrupted several times by the spy program at some points during the encryption.

SBox T_4 (the last set of 16 points in each row) plays a particular role here, as it is invoked only on the last round. Therefore, the SBox T_4 accesses indicate the end of an encryption, and all lines within the SBox T_4 accesses are then linked to a single encryption.

V.6. Analysis of the last round

Previously cited attacks use the information about all cache accesses of one encryption. However, we focus here on the accesses of the last round. Indeed, if the time resolution of the spy process enables us to see the accesses of one encryption, SBox T_4 will also appear clearly. Moreover, the last round analysis take advantage of the non-linearity of the `SubBytes` function; this particularity increases the performance of the attack.

The ciphertext is now under investigation in order to take advantage of the last round accesses. Recall from our introduction, the last round of AES is particularly of interest in the sense that the `MixColumns` operation is never applied. And for that particular reason, OpenSSL uses SBox T_4, especially for the last round. With $c := E_{AES}(p, k)$ being the ciphertext, we have the following relations linking c and the last round:

$$c = K^{(10)} \oplus \texttt{ShiftRows}[\texttt{SubBytes}[x^{(9)}]],$$

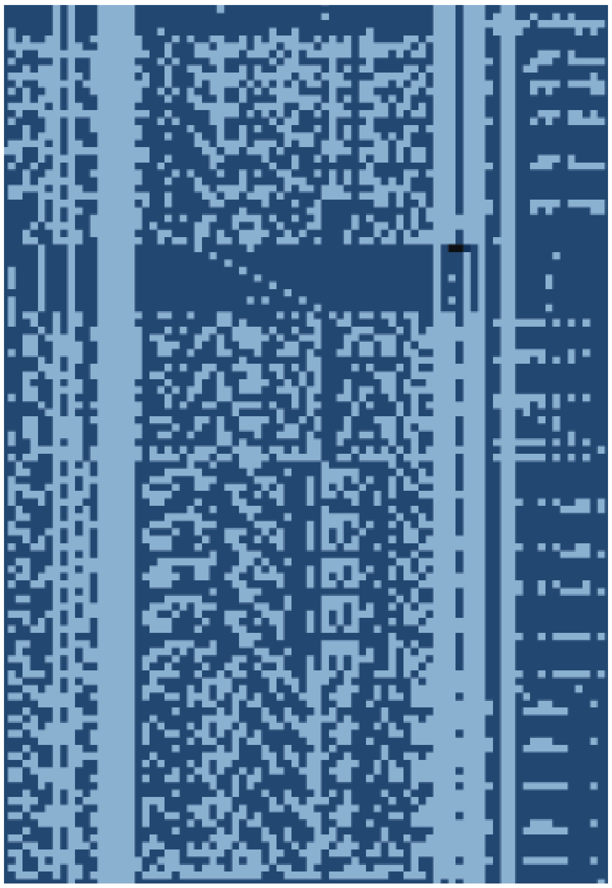


Figure 2: Evolution of the cache *versus* time, displaying several AES encryptions. Each horizontal line represents the state of the cache lines (represented by a point) at a given time. The brighter, the longer the time to access its corresponding cache line.

where $x^{(9)}$ is the initial state of round 10 (*i.e.* the output of round 9 and input to round 10). Since round 10 uses SBox T_4, we denote the actual access to T_4 by [SBox T_4 outputs]. The relation becomes:

$$c = K^{(10)} \oplus [\text{SBox } T_4 \text{ outputs}].$$

Therefore, we derive a relation defining $K^{(10)}$:

$$K^{(10)} = c \oplus [\text{SBox } T_4 \text{ outputs}],$$

from which it is easy to deduce the value of k from $K^{(10)}$ — cf. Chapter I.

However, [SBox T_4 outputs] represents the result of *all* the accesses of the last round, *i.e.* for all bytes of the T_4 input $x^{(9)}$. Moreover, the cache accesses only point out the accessed cache lines, but not the individual elements in those lines. Although the next section will give more details on the last points, we refer the reader to Handy [38] for a thorough review of cache architectures.

V.7. Average number of accesses for the last round

Let us first introduce some notations. Let $\delta = 2^o$ be the cache line size (in byte) and $m = 2^l$ be the number of cache lines of SBox T_4. Let also $p(b)$ be the probability that one specific cache line is accessed in b T_4 accesses, and $P(b)$ its corresponding random variable. Likewise, $p_n(b)$ the probability that one specific cache line is *not* accessed during b T_4 accesses, and $P_n(b)$ its corresponding random variable. Also, let us assume that the accesses to T_4 are independent and uniformly distributed. We now want to compute the expected number of different cache accesses into T_4. Using that $p(1) = 1/m$, or $p_n(1) = 1 - 1/m$ and the last assumption yields

$$p_n(16) = \left(1 - \frac{1}{m}\right)^{16}.$$

Therefore, the expected number of cache lines *not* accessed in a last round is given by

$$\mathbb{E}(P_n(16)) = 16 \cdot \left(1 - \frac{1}{m}\right)^{16}.$$

In the case of caches with 64 bytes per cache line (*i.e.* $\delta = m = 16$), we get $\mathbb{E}(P_n(16)) = 5.70$ and thus $\mathbb{E}(P(16)) = 10.30$ as the expected number of cache lines accessed during a last round.

V.8. Resolution

On Figure 2, the T_4 accesses are all visible within a few vertical lines. Let the resolution factor t be defined as

$$t := \frac{\#\text{ of ciphertexts}}{\#\text{ of measurements}},$$

which yields the following different resolution cases:

- *low resolution*: One measurement covers t encryptions (with $t > 1$) and therefore several last round accesses are overlaid. Then $\mathbb{E}(P(t \cdot 16)) = 16 \cdot (1 - 1/m)^{t \cdot 16}$. Table 1 shows that $\mathbb{E}(P(t \cdot 16))$ rapidly gets close to its limits.
- *one line resolution*: The frequency of measurements isolates one last round per measurement, *i.e.* $t = 1$. We already computed this case. Then $\mathbb{E}(P(16))$ equals 10.30 for $m = 16$.

t	2	3	4	5	6	7	> 7
$\mathbb{E}(P(t \cdot 16))$	13.97	15.28	15.74	15.91	15.97	15.99	≈ 16

Table 1: Expected number of cache lines of SBox T_4 accessed in t last rounds for $t > 1$ and $m = 16$.

- *high resolution*: There are several measurements $(1/t)$ occurring during the last round, *i.e.* $t < 1$. The observation of the evolution of the accesses gives a notion of the order in which the accesses have taken place and therefore narrows down the possible accesses per byte.

For now, we consider *one line resolution* to detail the analysis of the accesses. We return to this in Section V.10 and discuss the impact of the resolution in the analysis's results.

V.9. Non-elimination and elimination methods

We detail here how to deduce the secret key from cache accesses of SBox T_4 and the ciphertexts. We present in the following our non-elimination and elimination attacks.

The first method is directly inferred from the relation obtained above:

$$\mathbf{K}^{(10)} = \mathbf{c} \oplus [\text{SBox } T_4 \text{ outputs}].$$

This states that $\mathbf{K}^{(10)}$ is computed with the ciphertext $\mathbf{c}$ and *some* SBox outputs resulting from the SBox T_4 accesses. Each access to a particular line outputs one out of 16 values and we try to discover which one it is, from many ciphertext/accesses pairs. This finally leads to the value of $\mathbf{K}^{(10)}$, when applied in a byte-wise fashion.

The second method is based on the inverse relation:

$$\mathbf{K}^{(10)} \neq \mathbf{c} \oplus \neg[\text{SBox } T_4 \text{ outputs}],$$

where $\neg$[SBox T_4 outputs] refers to the non-accessed cache lines. The relation simply means that the bytes obtained by the addition of $\mathbf{c}$ and the non-accessed cache lines can be discarded as candidates for $\mathbf{K}^{(10)}$. This method, as we are about to see, requires less ciphertext/accesses pairs than the first one.

Let us call those methods respectively *Non-elimination* and *Elimination* methods, since they share the same philosophy as Tsunoo's methods [122]. Let us further suppose that we have a large number of clear measurements of the cache accesses over the last round and the corresponding ciphertexts. We will now detail each method individually.

V.9.1. Non-elimination method

This method is separated into three steps. All three steps must be applied for all of the 16 bytes of the key. Suppose we attack byte i, $0 \leq i \leq 15$.

(1) *Selection of the ciphertext*: The ciphertext/accesses pairs are sorted according to the value of byte i of the ciphertext. Since the key is constant, it is clear that if the ith byte of different ciphertexts have the same value, all the accesses corresponding to those ciphertexts must contain an access to one common cache line[1].

Consider for example Figure 3 as the accesses for a constant value of byte i (say `x00`).

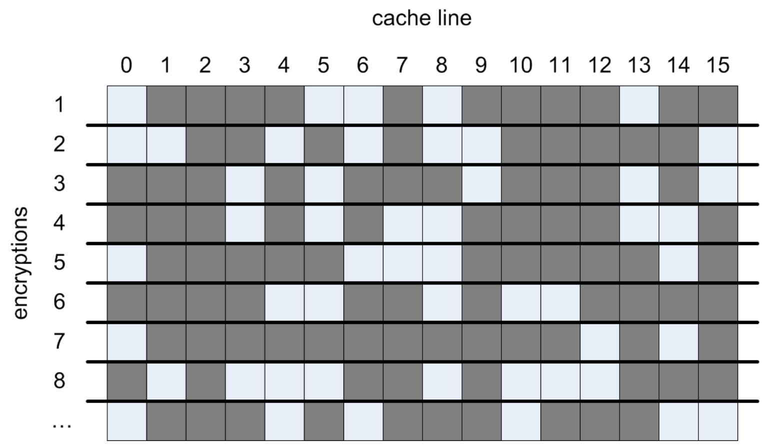


Figure 3: Cache line accesses for ciphertexts with a constant value for byte i. The dark boxes represent accessed cache lines.

(2) *Discovery of the correct access*: The access corresponding to the value of byte i is found by taking the unique access present on *every* encryption (cf. Figure 4 where byte $i =$ `x00` is found to be linked to cache line 2. We define in this case false positives as the wrong candidates present along with the correct candidate: *e.g.* the number of false positives on this example is

- 10 on encryption 1,
- 6 on encryption 2,
- 5 on encryption 3,
- 3 on encryptions 4 and 5,
- 1 on encryption 6 and further

[1]Since the ciphertexts can be considered random, the other bytes will have random accesses to T_4. We seek the constant access among the random ones.

- 0 on encryption 7 and further.

The probability of a false positive accessed for k successive encryptions is $\left(1-((m-1)/m)^{15}\right)^k$ and this gives less than 4 percents when $k = 7$.

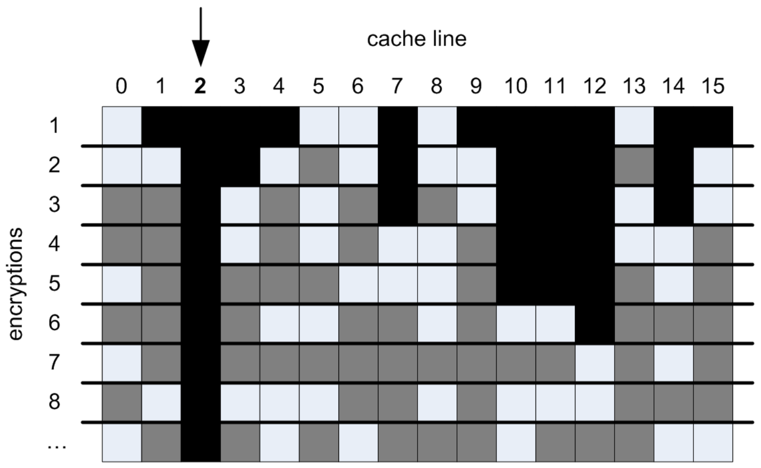


Figure 4: Highlight of the constant access. The dark boxes represent accessed cache lines and the black boxes show the evolution of the possible candidates.

(3) *Application of the difference*: The bitwise difference of the selected values of byte i must also link two elements in the corresponding access of T_4. Operation (2) showed that byte i = `x00` is linked to cache line 2. Let us assume that the same operation was being executed on a different value of byte i (*e.g.* `x01`) and the corresponding cache line was 5. Therefore the bitwise difference of the values for byte i is `x00` $\oplus$ `x01` = `x01` = 1. Hence we only need to find, in the lines 2 and 5 of SBox T_4, output values presenting the same difference. The two lines are shown below:

```
...
2        b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
...
5        53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
...
```

The only pair having a bitwise difference of 1 are `xfd` and `xfc`, when byte i is equal to respectively `x00` and `x01`. Therefore, the byte of $\mathbf{K}^{(10)}$ corresponding to byte i has a value of `xfd` $\oplus$ `x00` = `xfc` $\oplus$ `x01` = `xfd`. In the unlikely event of more than one match, the operation (2) must be repeated to establish other bitwise differences.

The expected number of pairs to find the correct key byte is

$$\sum_{n=2}^{\infty} p_{fp}(n) \cdot N(n) \approx 186,$$

with $p_{fp}(n)$ and $N(n)$ respectively being the probability of having a false positive after n pairs and the average number of pairs necessary to get two values repeated. The other bytes of $\mathbf{K}^{(10)}$ are found the same way, by considering another byte number.

V.9.2. Elimination method

Here, all bytes can be treated at the same time. We consider the case of byte i for the sake of clarity; it is straightforward to apply the method to the other ones. Let $\mathcal{V}$ be the set of all possible key byte values. Initially, $\mathcal{V}$ is composed of all 256 values a byte can take: $\mathcal{V} = \{j : 0 \leq j \leq 255 | j\}$. At the end, we want that $\mathcal{V} = \{k_i^{(10)}\}$. Consider for example that the ciphertext's byte i c_i equals `x2c` and the corresponding accesses are the ones displayed in Figure 5.

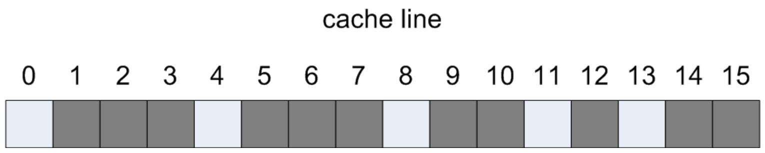


Figure 5: Example of accessed cache lines. The dark boxes represent accessed cache lines.

The accessed cache lines are

$$1\ 2\ 3\quad 5\ 6\ 7\quad 9\ 10\quad 12\quad 14\ 15$$

and the non-accessed ones are

$$0\ 4\ 8\ 11\ 13.$$

This method focuses on the latter list of cache lines. Let $\widetilde{\mathcal{A}}$ represent this subset of the cache lines and $n_{\widetilde{\mathcal{A}}}$ be the number of elements of $\widetilde{\mathcal{A}}$:

$$\widetilde{\mathcal{A}} = \{0, 4, 8, 11, 13\}, \qquad n_{\widetilde{\mathcal{A}}} = \left|\widetilde{\mathcal{A}}\right| = 5.$$

By the elimination relation, $\mathbf{K}^{(10)} \neq \mathbf{c} \oplus \neg[T_4 \text{ outputs}]$, each non-accessed cache line enables us to remove all key candidates corresponding to this access. In our example, this means that for the first element of $\widetilde{\mathcal{A}}$ we have:

$$\begin{aligned}K_i^{(10)} &\neq c \oplus [\text{cache line } 0]\\ &\neq \texttt{x}\,\texttt{2c} \oplus {}_{\texttt{x}}\{\texttt{63,7c,77,7b,f2,6b,6f,c5,...,2b,fe,d7,ab,76}\}\\ &\neq {}_{\texttt{x}}\{\texttt{4f,50,5b,57,de,47,43,e9,...,07,d2,fb,87,5a}\} = \mathcal{V}_e,\end{aligned}$$

where $\texttt{x}\cdot\cdot$ and ${}_{\texttt{x}}\{\cdot\cdot\}$ represent as previously hexadecimal values. All values of $\mathcal{V}_e$ can then be eliminated from $\mathcal{V}$:

$$\mathcal{V} \leftarrow \{j : 0 \leq j \leq 255\} \backslash \mathcal{V}_e$$

Then we go to the next element of $\widetilde{\mathcal{A}}$ (*i.e.* cache line 4) and apply the same technique. The cache line bytes are

$${}_{\texttt{x}}\{\texttt{09,83,2c,1a,1b,6e,5a,a0,52,3b,d6,b3,29,e3,2f,84}\}$$

added with $\texttt{x}\,\texttt{2c}$ gives new candidates to eliminate. Then $\mathcal{V}$ is updated as:

$$\mathcal{V} \leftarrow \mathcal{V} \backslash \{\texttt{25,af,00,36,37,42,76,8c,7e,17,fa,9f,05,cf,03,a8}\}.$$

This is then repeated for the three other cache lines in $\widetilde{\mathcal{A}}$.

For one given ciphertext/accesses pair, each cache line ends up eliminating 16 different values from the byte candidates: the ciphertext byte is constant and the SBox outputs are all different from each other. In our example 80 candidates have been eliminated with the pair under consideration. Then, another ciphertext/access pair is analyzed and the same technique is applied with the non-accessed cache lines of that pair. The ciphertext's byte i and the non-accessed cache lines are probably different from the previous analyzed pair. Therefore the subset of wrong candidates deduced from this pair should only present a few collisions with the one of previous pairs.

However, there will be more and more collisions as the number of wrong key candidates becomes closer to one. Consider for example the following table showing the reduction of the number of wrong key candidates, for a practical case.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\lvert\mathcal{V}\rvert$	255	175	119	77	55	33	21	15	6	6	2	2	2	1	1	0

Then other pairs are analyzed until there is only one key byte remaining[2]. It is important to note that this method allows to work on all the bytes, at the same time. In this case, we need 16 subsets (one per byte) keeping the count of all candidates:

$$\mathcal{V}_0, \ldots, \mathcal{V}_{15}.$$

[2]Note that one can stop anytime and run an exhaustive search on the remaining key byte candidates.

Let us first compute the number of pairs needed to distinguish the right key candidate, for one byte. We can now define s, the number of candidates eliminated by the analysis of one ciphertext/access pair. At the beginning, we have 255 wrong candidates. With the first pair, we eliminate s of them and the number of wrong candidates is 255-s. However, starting at the second pair, collisions can occur. Therefore, we approximate the number of remaining wrong key candidates after n pairs by $255 \cdot (1 - s/255)^n$. This formula is validated by simulation with $s = 16$ (see Figure 6).

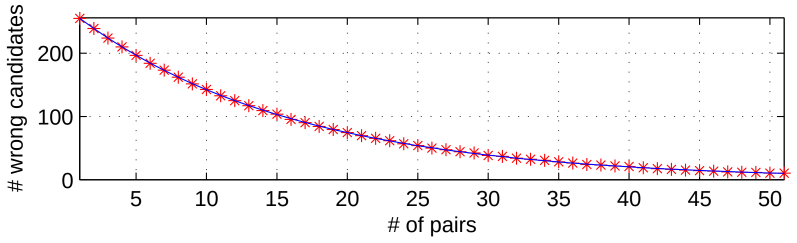


Figure 6: Experimental verification of the reduction formula, yielding the number of wrong key candidates per pairs. The continuous line is the theoretical formula and the stars are the simulated data.

We then apply the formula with the expected number of non-accessed cache lines (*i.e.* 5.70, for 64-byte cache lines). Hence, substituting $s = 5.70 \cdot 16$ into $255 \cdot (1 - s/255)^n$ we get the following results[3] (cf. Table 2).

# pairs	$\lvert\mathcal{V}\rvert$	# pairs	$\lvert\mathcal{V}\rvert$
0	255	8	7
1	164	9	5
2	105	10	3
3	68	11	2
4	43	12	1
5	28	13	0.8
6	18	14	0.5
7	12	15	0.3

Table 2: Theoretical results of wrong key candidates, per pair ciphertext/accesses, for $m = 5.70 \cdot 16$.

After approximatively 14 pairs, the number of wrong candidates for one byte should be close to 0. This shows that less than 20 ciphertext/accesses pairs are needed to recover the whole $\mathbf{K}^{(10)}$ subkey and

[3]The data differ from the ones in the practical case because there are 5 non-accessed cache lines whereas 5.70 are considered in Table 2.

therefore also the secret key $\mathbf{k}$. As the analysis methods (elimination and non-elimination) are simple deductions, they can easily be done in an extremely low computation time (under a second). However, the elimination method gives a much better performance than the non-elimination one, in term of numbers of measurements required to break a full 128-bit AES key.

V.10. Practical considerations

Let us now re-elaborate the question of the measurement resolution.

- *Low resolution*: Table 1 highlighted that the expected number of accessed cache lines rapidly approaches to 16, when the number of encryptions between two measurements increases. However, even if the leakage gets smaller, every ciphertext/accesses pair with at least one non-accessed cache line carries information. Moreover, low resolution implies multiple ciphertexts for a single cache information (*i.e.* one line combines all the accesses corresponding to the ciphertexts). In this case the analysis must integrate the different possible ciphertext values and statistically derive the most likely key bytes.
- *One line resolution*: As detailed above, 5.70 cache lines are not accessed. The analysis does not need to deal neither with the multiple ciphertexts issue nor with the order inside the accesses.
- *High resolution*: Both methods are still possible. But the leakage also gives some information about the order of the accesses. One can then increase the performances of the analysis and therefore reduce the number of required pairs, by correlation of the byte accesses in the AES program and the accesses visible in the measurements. For $t \leq 1/16$, one can clearly identify the byte accesses. Two to three pairs only makes it possible to find the correct candidates for all key bytes[4]. High-resolution would also make possible theoretical attacks on the key expansion algorithm, although it is harder for an outsider to trigger a key expansion than an encryption.

Finally, we considered here that the cache accesses were exempted of any measurements noise. However practical attacks must deal with noise in the measurements. Consider for example Figure 2 which presents vertical stripes and a diagonal line in the upper half. The presence of noise in the measurements increases the number of accessed cache lines.

[4]The elimination and non-elimination methods then presents the same performances.

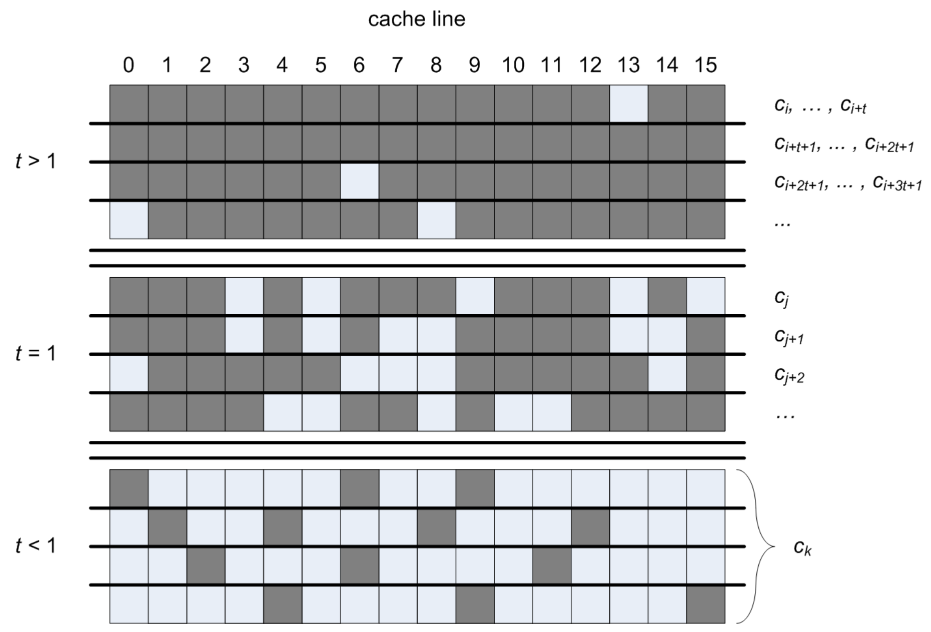


Figure 7: Different resolutions for access-driven cache-based attacks. The resolution factor t defines the ratio *# of ciphertexts / # of measurements*.

However, the techniques that we detailed here can still be exploited, by taking into account the noise[5].

We gave above the minimum expected number of measurements to perform the attacks for $t = 1$. As this boundary is precious and has been practically confirmed it should used to evaluate the efficiency and security of current software implementations which are hardened by corresponding countermeasures.

V.11. Other attacks

In [7, 8], Bertoni *et al.* developed an attack based on the processor's power consumption. However, their experiments were conducted on a MIPS-like processor using the Simplescalar-3.0 toolset (software emulator, see [18]). Analyzing the simulated power traces enabled them to easily detect cache-misses from cache-hits, by analyzing longer and higher consumption peaks. Nevertheless those papers are lacking practical considerations on real processors.

[5]Also, the location of the vertical stripes is variable between different runs of the setup.

Other recent works on cache vulnerabilities have been performed. The particularity of those attacks is the leakage. They are indeed based on the assumption that it is possible to capture the sequence of cache-hit and -miss for the SBox accesses. Those attacks are therefore called *trace-driven attacks*.

C. Lauradoux detailed this idea in [56]. Based on the assumption that a processor's power traces can be measured (and carry some information, as claimed by Bertoni *et al.* [7, 8]), he explained how to treat the sequences of cache-hits and -misses (resp. H and M) of the first round:

$$M\ M\ M\ M\ H\ M\ H\ M\ M\ M\ H\ M\ M\ H\ M\ M.$$

He deduced relations between key bytes that finally reduce the exhaustive search complexity from 2^{128} to

- 2^{80} with 16 encryptions of an AES implementation involving large precomputed SBox tables;
- 2^{68} with 240 encryptions of an AES implementation involving a single SBox tables

on a 64-byte cache line processor.

A recent report [1] reformulates this idea and provides estimations of the cost of such attacks with noiseless measurements.

V.12. Summary

We surveyed in this Chapter a type of side channel that takes advantage of the sharing of the memory architecture between processes, on most of the processors. One process is indeed able to access data of any size and thereby it loads the cache. Hence, other processes' data are evicted to another level of the memory architecture. And when these other processes regain access to the processor, the evicted data are brought back with higher delays than for not evicted ones. This simple mechanism enables a process to spy on another's activity through the time measured to re-access its (previously loaded) data.

The spying reveals informations about the memory accesses of the other process, but does not directly leak the content of the other process's data.

Our contribution is two-fold: we detailed a software method to achieve snapshots of cache accesses on single-threaded processors and we showed that the analysis of the last round of AES enables the full disclosure of an 128-bit AES key with less than 20 encryptions. Where previous studies focused exclusively on a minority of processors, we investigated the access-driven cache-based attacks on single-threaded processors. We explained our strategy and why it is solely depending on software engineering. Moreover, we chose the challenging case of AES: its short execution time (compared to RSA's) demonstrates the fine granularity of our cache accesses' snapshots. Our software strategy can easily be adapted and combined with previously reported access-driven attacks on any single-threaded processor. Moreover, on common implementations the last round is performed with the help of a special precomputed table. Through this feature, we achieved to infer more information than with other strategies. We gave expected numbers of measurements, depending on the granularity and noise of the access-driven measurements. This contribution sets new boundaries for countermeasures against cache-based attacks. For example, some software mitigations proposed to apply masking techniques and to renew the mask every 256 encryptions. With respect to the described methods, we showed in this Chapter why this number should be reconsidered.

Further readings

[82] *Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In David Pointcheval, editor,* CT-RSA*, volume 3860 of* Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.*

[88] *Colin Percival. Cache missing for fun and profit, 2005. Available online at* `http://www.daemonology.net/hyperthreading-considered-harmful/`.

CHAPTER VI

Mitigations of Cache-based Side Channels on AES

Abstract: Hardware side channel vulnerabilities have been a center of attention for over a decade in the smart card industry. Recent developments have diverted a part of the side channel concerns to the PC world. In particular the behavior of the processor's cache leaks information and a software can collect and analyze those side channels. Whereas the attacks have been largely disseminated and the source of the leakage is mostly understood, few countermeasures have been given. In this Chapter, we detail the software countermeasures for the case of AES. They can be adapted to match a security level with respect to a given thread model: higher security comes at the price of more consumed resources. This contribution provides the first AES codes mitigating cache-based side channels.

VI.1. Introduction

As shown in previous chapters, cache-based attacks are deeply linked to the very essence of the processor's memory architecture. The cache mechanism being controlled (1) internally by the processor and (2) transparently for the processes, one of them cannot bypass the cache, *i.e.* the cache mechanism is an automatic process (not controlled by the CPU nor by the OS) and therefore the data will be ended up in the cache after each reference to them. Similarly, a process cannot prevent the processor from switching to another process nor advising it on a suitable time to perform the switch[1]. And this other process can infer information about the first one, because the cache is not cleared during the switches for efficiency reasons.

On the other hand, an increasing number of private and commercial activities requires security-related applications to run on standard PC

[1]Evenmore the CPU has no *consciousness* or idea that processes are switched. Context switches are series of stores of the current CPU state and loads of a previous one.

platforms and the threat of hardware-based[2] side channels requires immediate attention. Actions can be taken at three levels:

- **Operating System**: The OS being the computer's conductor, it manages the hardware - software interfaces, controls and allocates the memory and decides the ordering of the processes. It therefore has a role to play in mitigating hardware-based leakages and side channel exploits. Here are some examples among others:
  - *Address Space (Layout) Randomization* (ASR or ASLR) is a technique to prevent buffer overflow attacks by randomizing the memory location of specific system components (position of libraries, heap and stack) [4, 10, 32, 105]. It is implemented in several OS as OpenBSD [26] and Linux [117, 126][3]. Experiments of Bernstein's time-driven attack were unsuccessful on OS with ASLR, probably because of the randomization of the crypto library. It is however unclear whether ASLR could prevent access-driven attacks. Nonetheless, attacks have demonstrated that it is possible to find critical memory zones randomized by ASLR [105] within 5 minutes. This failure is principally due to the low number of randomization bits available on 32-bit architectures. Address randomization techniques [4, 9, 14, 50] can make cache-based attacks harder to mount *if and only if* the randomization is updated frequently enough to prevent an attacker from localizing the SBox Tables' location. A thorough analysis is required to evaluate and quantify the performance of randomization techniques against cache-based attacks. This is out of the scope of the present dissertation.
  - *Cache control* could be provided by the OS to transparently reload the cache content for programs requesting this feature. Security-tagged programs would not leak from other processes' eviction if all their data are loaded in the cache before the process is executed. This technique requires attention from the software programmers. However, this does not prevent evictions between data inside the security process (self-eviction) that could generate timing variabilities.

[2]Cache-based attacks are the first side channel hitting the processor industries with such impact. But other parts of the processor are leaking information; other attacks are yet to come. See comments in Chapter VIII.

[3]Note in Windows XP Pro and 2003, ASR stands for Automatic System Recovery; this has no connection with the ASLR techniques mentioned here.

Even if some OS-based features already could mitigate cache attacks, this field requires more efforts to prove the effectiveness of its countermeasures.

- **Hardware**: As cache-based attacks are based on hardware features, it sounds natural to find hardware solutions. However, there is no obvious solution that does not deteriorate the performance of CPU architectures. Although the cache system is pointed out as the problem, this crucial part is the keystone of the processor, providing a way to hide the real latency of its memory to the CPU. Disabling components, would it be possible, is not an option. An acceptable solution would be highly dependent on the processor and on the application. Nonetheless, some possible approaches are given here:
  - *Hardware crypto-coprocessor* is a common feature in smart cards and security tokens. Ten years of side channel analysis on those objects have provided the industry and the academia with a strong expertise in hardware crypto engines. Even if the usage model behind smart cards is different than the one of general-purpose microprocessors, a large part of the know-how can be directly transfered to build efficient and secure implementations of on-core crypto-coprocessors.
    Moreover, many other sectors already make an intensive utilization of hardware accelerators and their experience can be helpful. The requirements in terms of cryptographic algorithm should be carefully studied. Nevertheless, considering the wide variety of applications using AES, it would probably be one of the first to be integrated.
  - *Logical operations* can replace the SBox Table lookups. It would represent a smaller hardware overhead than a full crypto-accelerator. This can provide rapid and constant-time operations. They are however less efficient than direct SBox Table lookups.
  - *Cache modification* can also be imagined as countermeasures. Several methods have been proposed [82, 86]. However, modifying the cache mechanism is intimately linked to a specific CPU architecture and viable solutions must be considered for each instance.

  Even if they appear to potentially provide low performance costs, it would be unexpected to see any hardware-based countermeasures appearing in short terms into general-purpose microprocessors, due to the long silicon integration time.
- **Software**: Conversely to OS and hardware, software mitigations a greater reactivity to response to security threads. Patches are indeed rapidly written, tested and deployed, with

generally an overhead in execution time and/or memory requirements.

This Chapter deals with this last category of mitigations. In the particular case of cache-based side channels, the objective of the proposed countermeasures is to provide a panel of solutions with different levels of security. However, the more secure, the more resource-consuming. A thorough thread analysis must be made in order to select the appropriate security level with the minimal overhead.

Our software mitigation strategy is based on three principles: (1) compact tables, (2) frequently randomized tables and (3) pre-loading of relevant cache-lines. Using all three countermeasures simultaneously in an efficient manner and individually scaling them appropriately for the power of the side channel adversary, sets our mitigation strategy apart from all previously proposed mitigations.

VI.2. Discussion of a side channel adversarial threat model

To study the effectiveness of our mitigation strategies, we now introduce the notion of a side channel adversarial threat model. We do this in order to describe the effectiveness of the mitigations separately from discussing the effectiveness of an adversary's ability to exploit a software side channel vulnerability. In a software side channel, the adversary is executing a spy process on a platform that is also executing a crypto process in a multi-tasking environment. To study the cache side channel, the important ingredient is the accuracy of the information that the adversary can obtain about the cache accesses of the target process. Thus, as we describe mitigations, we discuss how frequently the spy process would need to get data about the cache accesses of the target process in order to defeat the mitigation.

According to Chapter V, for AES the spy process is trying to obtain information about which cache lines are being accessed by the crypto process when the crypto process accesses the S-Box tables. Against an adversary who is able to obtain precise information about all cache accesses, a mitigation for the crypto process would be to access all cache lines of the tables each time it needed to access any entry in the table. This would be quite inefficient. However, this is probably a higher bar than is really necessary, since we do not know of any method for a spy process to obtain this precise information. Thus, we present several different alternate methods for defending against more realistic (and demonstrated) spy processes.

The *first* method we present involves using a compact S-Box table which fills (on most modern general-purpose processors, *i.e.* with 64-byte cache lines) only 4 cache lines, and then accessing each of the 4 cache lines in every round. This method will defend against an adversary who is not able to observe cache access behavior more frequently than the time required by the crypto process to execute an AES round (refer to [6, 121]).

The *second* method we present does the above process only for the first and last rounds, but for the middle rounds (2 through 9), uses even larger S-box tables (which is more efficient). The reason this may still be an effective mitigation is because it is more difficult for the adversary to use information about the cache accesses of the middle rounds. Instead of accessing every cache line of these larger tables with every round, we permute the tables for some number of encryption blocks, thus further obscuring the information from cache accesses. We conjecture that this method will effectively defend against an adversary who is only able to obtain information about cache access behavior every few rounds. This is perhaps conservative, since we are not aware of any attack that would be effective against this method by an adversary who was able to obtain information for individual AES rounds.

Finally, we introduce a *third* method that may be an effective mitigation even against an adversary who is able to observe cache access behavior multiple times during a round of AES. This method also uses the compact S-Box Tables, and accesses each cache line in the table at every round, but adds the additional step of permuting the compact S-Box Tables, and changing the permutation very frequently.

In the remainder of this Chapter, we give more precise descriptions and security arguments for these mitigation methods, and discuss their performance on experimental implementations.

VI.3. Mitigations against cache-based side channel attacks

As aforementioned we now explain our three individual mitigation strategies in more detail: (1) compact S-box table, (2) frequently randomized tables, (3) pre-loading of relevant cache-lines. However, for the reason of interest we present our mitigations in a more compact form. Namely, we will present the use of the compact S-box (first and last round) and the large-table based (inner) rounds only while making direct use of a permutation P. Therefore, assuming that a permutation P is somehow given and can be efficiently computed, we first show how a permuted compact S-box

$$\texttt{byte} \quad \texttt{PinvS}[0:255]$$

as well as a large permuted table

$$\texttt{word} \quad \texttt{T}[0:511]$$

can be efficiently computed.

VI.3.1. Constructing permuted S-box and inner round T table.

For later use, we show in Figure 1 how to compute the randomized (via P) compact S-box table `PinvS` as well as the randomized (via P) large table `T`.

```
Input: Permutation $P$ (with parameters $A$ and $B$)
Output: Permuted S-box PinvS[0:255] and inner round table T[0:511]

  S-box is a 256-byte vector S[0 : 255];
  (byte [0 : 255], word [0 : 511]) function permute_S_box(S[0 : 255], $P$);
  begin
    byte PinvS[0 : 255], m;
    word T[0 : 511];

    for all $i$ do parallel PinvS[$i$] := 0;
    for all $j$ cache_blocks in S do
    begin
      for all $i$ entries in S do
      begin
        $p := P(i)$;
        $m :=$ if offset $p$ is in cache block $j$ then 255 else 0;
        PinvS[$j * cacheblocksize + p \bmod cacheblocksize$] : + = $m \wedge$ S[$i$];
      end
    end
    /* for version 2, build expanded table T for inner rounds from PinvS */
    for all $i \in [0 : 255]$ do
    begin
      word v1, v2, v3, c;

      v1 := PinvS[$i$];
      c := if v1 > 127 then $0x1b$ else 0;
      v2 := (v1 $*$ 2) $\oplus$ c mod 256;
      v3 := v1 $\oplus$ v2;
      /* store twice so we can implement rotation by unaligned load */
      T[$2*i$] := T[$2*i+1$] := v2 $\oplus$ (v1 <<< 8) $\oplus$ (v1 <<< 16) $\oplus$ (v3 <<< 24);
    end
    return (PinvS, T);
  end
```

Figure 1: Constructing permuted S-box and inner round `T` table.

VI.3.2. Permuted compact round

First, we show how to efficiently use a single and permuted compact S-box table (256 bytes-table) in a *single* AES round. As already pointed out by [82] (and easily derivable from our Chapter V), substituting the five 1KB Tables (as used in OpenSSL) by a compact S-box Table (256 bytes-table) dramatically reduces the information leakage due to potential cache misses.

But, in contrast to [82] we add additional protection mechanisms to the use of a single compact table:

- As the 256-byte Table fits in only 4 cache lines, we can now afford to pre-fetch all 4 cache-lines of the compact Table into the cache prior to using the Table.
- Frequently permuting the compacted Table with a new permutation injects lots of entropy against adversaries which are relying on statistical success probabilities.

The following Figure 2 shows a permuted compact round with prefetching of relevant cache-lines. The corresponding x86-assembler program is concluded in Section VI.5. Also, thanks to the x86 SSE SIMD instructions, we observe that the security-critical computations are totally constant in time and are all done in parallel without any branches.

```
Input: 16-byte vector b[0 : 15]
Output: the result of applying one AES-round transformation to b

  permuted S-box is a 256-byte vector PinvS[0 : 255];
  as permutation $P$ we simply implemented $P(x) := (B + x) * A \bmod 256$
    where A is odd;
  /* $P$ can be efficiently computed with SSE instructions */
  $r$ is a permutation of 16 bytes $(0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11)$

  byte [0 : 15] function compact_encrypt_round(b[0 : 15], roundkey[0 : 15], $A$, $B$);
  begin
    byte v1[0 : 15], v2[0 : 15], v3[0 : 15], c[0 : 15];

    for one representative $i$ from each cache block of S, touch S[$i$];
    for all $i$ do b[$i$] := (b[$i$] + $B$) * $A$ mod 256;
    for all $i$ do v1[$r[i]$] := PinvS[$i$];
    /* complete linear transform with SSE SIMD-instructions */
    for all $i$ do parallel c[$i$] := if v1[$i$] > 127 then $0x1b$ else 0;
    for all $i$ do parallel v2[$i$] := (v1[$i$] * 2) $\oplus$ c mod 256;
    for all $i$ do parallel v3[$i$] := v1[$i$] $\oplus$ v2[$i$];
    return roundkey $\oplus$ v2 $\oplus$ blrm4(v1, 1) $\oplus$ blrm4(v1, 2) $\oplus$ blrm4(v3, 3);
  end

  byte [0 : 15] function lbrm4(v[0 : 15], $j$);
  begin
    byte t[0 : 15];

    for all $i$ do parallel t[$4 * \lfloor (i/4) \rfloor + ((i \bmod 4 + j) \bmod 4)$] := v[$i$];
    return t;
  end
```

Figure 2: Permuted compacted round.

VI.3.3. Permuted non-compact tables

After having seen how to permute (via permuation P) a compact round, we will show now how to compute the inner rounds by using the large permuted table T from Figure 1. The following Figure 3 shows the corresponding pseudo-code, while the corresponding x86-assembler program is concluded in Section VI.5.

```
Input: 16-byte vector b[0 : 15]
Ooutput: the result of applying one AES-round transformation to b

  S-box is a 256-byte vector S[0 : 255];
  T is a 2048-byte table for AES's linear transform and PinvS
    to enable unaligned rotates

  byte [0 : 15] function inner_round_encrypt(b[0 : 15], roundkey[0 : 15]);
  begin
    byte t[0 : 15];
    word w[0 : 3];

    /* for performance we cannot touch all cache-lines in the inner rounds */
    for all $i$ do t[$i$] := (b[$i$] + $B$) $*$ $A$ mod 256;
    w[0] := Word(T, $8 * $t[0]) $\oplus$ Word(T, $3 + 8 * $t[5])$\oplus$
      Word(T, $2 + 8 * $t[10]) $\oplus$ Word(T, $1 + 8 * $t[15]) ;
    w[1] := Word(T, $8 * $t[4]) $\oplus$ Word(T, $3 + 8 * $t[9])$\oplus$
      Word(T, $2 + 8 * $t[14]) $\oplus$ Word(T, $1 + 8 * $t[3]) ;
    w[2] := Word(T, $8 * $t[8]) $\oplus$ Word(T, $3 + 8 * $t[13])$\oplus$
      Word(T, $2 + 8 * $t[2]) $\oplus$ Word(T, $1 + 8 * $t[7]) ;
    w[3] := Word(T, $8 * $t[12]) $\oplus$ Word(T, $3 + 8 * $t[1])$\oplus$
      Word(T, $2 + 8 * $t[6]) $\oplus$ Word(T, $1 + 8 * $t[11]) ;
  end

  word function Word($x, y$) $\equiv$
    extracts from byte array $x$ 4 bytes at byte offset $y$ and returns 32-bit word;

  byte [0 : 3] function Bytes($x$) $\equiv$
    returns a byte array for a given 32-bit word $x$;
```

Figure 3: Permuted non-compact round.

VI.4. Practical results of our mitigations

Although our aforementioned software mitigation strategies are very flexible and allow various combinations with different security and performance strengths, we only tested the following simple configurations.

V1: All rounds compact, no permutation.
V2: Outer rounds (round 1 and round 10) are compact and the inner rounds are all large (round 2 until round 9), and additional tables are periodically permuted.
V3: All rounds are compact and tables are periodically permuted.

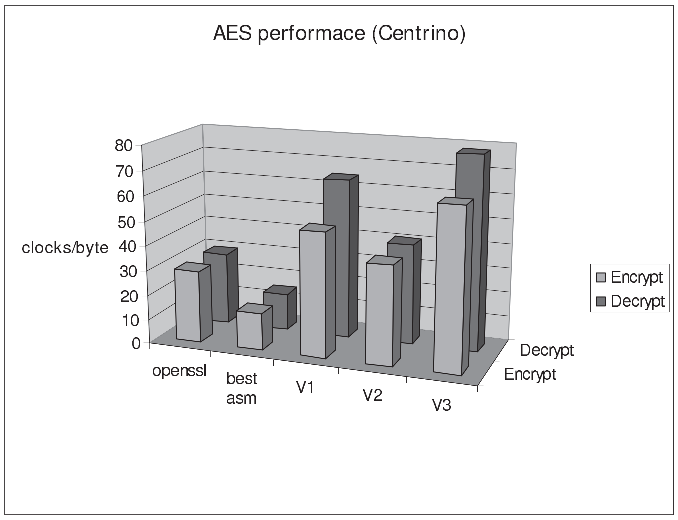


Figure 4: Performance of mitigations for AES.

VI.4.1. Performance

The following Figure 4 succinctly summarizes our performance results when comparing the above three variants with the performance of the OpenSSL and the best assembler implementation of the AES encryption/decryption algorithm.

VI.4.2. Security

We experimentally verified that all of the methods described above are effective in removing the timing vulnerabilities that were exploited by Bernstein [6]. This is easily visible from the following Figure 5. The execution times are averaged, regarding the value of one fixed byte of the plaintext, over a large number of random plaintexts (measurements have been taken according to [6] and Chapter IV). As expected, the distribution follows a Gaussian distribution.

VI.5. Mitigation codes

First and second round code for our mitigation version 2.

1st and last round code for V2

```
/* SSE version first round code--prefetch all 4 lines for outer rounds and use
compact S-box table with LT computation and does the permutation (x+B)*A */
            mov         edi,key                 // [ebp+16]
            mov         esi,inptr               // [ebp+8]
            movdqa      xmm7,SSEmask
            push        ebp
            movdqa      xmm6,[SSE\_MULT+edi]
```

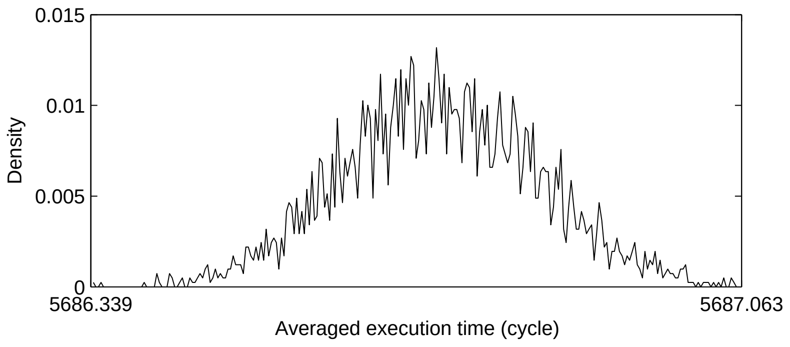


Figure 5: Distribution of execution times for our protected implementation (version V2) of AES.

```
            movdqa      xmm5,[SSE\_BIAS+edi]
            movdqu      xmm0,0[esi]
            pxor        xmm0,[SSE\_KEY\_OFFSET+0*16+edi]  // key addition
         // begin round 1
            paddb       xmm0,xmm5               // add bias B
            add         ebp,[SSE\_SBOX+edi]      // touch all sbox lines
            movdqa      xmm1,xmm7               // lower byte mask
            pandn       xmm1,xmm0  // pick off upper bytes (using not of mask)
            add         ebp,[SSE\_SBOX+64+edi]   // touch all sbox lines
            pmullw      xmm0,xmm6               // multiplier A
            add         ebp,[SSE\_SBOX+128+edi]  // touch all sbox lines
            pand        xmm0,xmm7               // lower byte mask
            pmullw      xmm1,xmm6      // note LSB is zero, and will stay zero
            add         ebp,[SSE\_SBOX+196+edi]  // touch all sbox lines
            por         xmm0,xmm1               // merge bytes back
            movd        eax,xmm0                // pextrw   eax,xmm0,0
            pextrw      ebx,xmm0,2
         // try unaligned word loads with masking
            movzx       esi,al                  // 0, 5, 10, 15
            movzx       ebp,[SSE\_SBOX+esi+edi]
            pextrw      ecx,xmm0,5              // load 10, 11
            movzx       esi,bh                  // 5
            mov         esi,[SSE\_SBOX-1+esi+edi]
            pextrw      edx,xmm0,7              // load 14, 15
            and         esi,0xff00
            or          ebp,esi
            movzx       esi,cl                  //  10
            mov         esi,[SSE\_SBOX-2+esi+edi]
            and         esi,0xff0000
            or          ebp,esi
            movzx       esi,dh                  // 15
            mov         esi,[SSE\_SBOX-3+esi+edi]
            and         esi,0xff000000
            or          ebp,esi
            movd        xmm2,ebp
            movzx       esi,ah //do parts of last row 1 before shifting eax,ecx
            mov         ebp,[SSE\_SBOX-1+esi+edi]
            and         ebp,0xff00
            movzx       esi,ch                  // 11
            mov         esi,[SSE\_SBOX-3+esi+edi]
            pextrw      ecx,xmm0,4  // pextrw   ecx,xmm2,4  // load 8,9
            and         esi,0xff000000
            or          ebp,esi
            movd        xmm4,ebp               // save partial result
            movzx       esi,bl                 // 4, 9, 14, 3
            shr         eax,16           // finish load 2, 3 from initial movd
            movzx       ebp,[SSE\_SBOX+esi+edi]
            movzx       esi,ch                 // 9
            mov         esi,[SSE\_SBOX-1+esi+edi]
            and         esi,0xff00
            or          ebp,esi
            movzx       esi,dl                 // 14
            mov         esi,[SSE\_SBOX-2+esi+edi]
            pextrw      ebx,xmm0,3             // load 6, 7
            and         esi,0xff0000
            or          ebp,esi
            movzx       esi,ah                 // 3
            mov         esi,[SSE\_SBOX-3+esi+edi]
            and         esi,0xff000000
            or          ebp,esi
            pextrw      edx,xmm0,6             // load 12, 13
            movd        xmm3,ebp
            movzx       esi,cl                 // 8, 13, 2, 7
            movzx       ebp,[SSE\_SBOX+esi+edi]
            unpcklps    xmm2,xmm3              // combine first two rows
            movzx       esi,dh                 // 13
            mov         esi,[SSE\_SBOX-1+esi+edi]
            and         esi,0xff00
            or          ebp,esi
            movzx       esi,al                 // 2
            mov         esi,[SSE\_SBOX-2+esi+edi]
            and         esi,0xff0000
            or          ebp,esi
            movzx       esi,bh                 // 7
            mov         esi,[SSE\_SBOX-3+esi+edi]
            and         esi,0xff000000
            or          ebp,esi
            movd        xmm3,ebp

         // finish last row
            movzx       esi,dl                 // 12, 1, 6, 11
            movzx       edx,[SSE\_SBOX+esi+edi]
            pxor        xmm1,xmm1              // preload zero
            movzx       esi,bl                 // 6
            mov         esi,[SSE\_SBOX-2+esi+edi]
            and         esi,0xff0000
            or          edx,esi
            movd        xmm0,edx
            por         xmm0,xmm4       // merge partial results for last row
            unpcklps    xmm3,xmm0       // combine 3rd and 4th rows
            movdqa      xmm0,SSEmask1B  // preload constant
            unpcklpd    xmm2,xmm3       // combine all rows
         // compact table linear transform
            pcmpgtb     xmm1,xmm2       // < 0 means top bit is on
            movdqa      xmm4,xmm2       // copy v1
            paddb       xmm2,xmm2
            pand        xmm1,xmm0       // masked load of 1b
            movdqa      xmm0,xmm4       // copy v1
            pxor        xmm2,xmm1       // v2
            movdqa      xmm1,xmm4       // copy v1
            psrld       xmm4,16         // v1 >> 16
            movdqa      xmm0,xmm1       // continue copying v1
            pslld       xmm1,8          // v1 << 8
            pxor        xmm0,xmm2       // v1 \^{} v2
            pxor        xmm2,xmm4       // xor in v1 >> 16
            psrld       xmm4,8          // continue shifting: v1 >> 24
            pxor        xmm2,xmm1       // xor in v1 << 8
            pslld       xmm1,8          // continue shifting: v1 << 16
            pxor        xmm2,xmm4       // xor in v1 >> 24
            movdqa      xmm4,xmm0       // copy v1\^{}v2
            pslld       xmm0,24         // (v1\^{}v2) << 24
            pxor        xmm2,xmm1       // xor in v1 << 24
            psrld       xmm4,8          // (v1\^{}v2) >> 8
            pxor        xmm2,xmm0       // xor in (v1\^{}v2) << 24
            pxor        xmm2,xmm4       // xor in (v1\^{}v2) >> 8
         // end of linear transform
            pxor        xmm2,[SSE\_KEY\_OFFSET+1*16+edi] // key addition
```

The inner round code for version 2 of our mitigation that uses the big (permuted) tables in the inner rounds.

Inner round code for V2

```
         // begin round 2
            paddb       xmm2,xmm5  // add bias B
            movdqa      xmm3,xmm7  // lower byte mask
            pandn       xmm3,xmm2  // pick off upper bytes (using not of mask)
            pmullw      xmm2,xmm6  // multiplier A
            pand        xmm2,xmm7  // lower byte mask
            pmullw      xmm3,xmm6  // note LSB is zero, and will stay zero
            por         xmm2,xmm3  // merge bytes back
            movd        eax,xmm2   // pextrw   eax,xmm2,0
            pextrw      ebx,xmm2,2
            movzx       esi,al     // 0
            mov         ebp,[SSE\_TABLES+8*esi+edi]
            pextrw      ecx,xmm2,5
            movzx       esi,bh     // 5
            xor         ebp,[SSE\_TABLES+3+8*esi+edi]
            pextrw      edx,xmm2,7
            movzx       esi,cl     // 10
            xor         ebp,[SSE\_TABLES+2+8*esi+edi]
            movzx       esi,dh     // 15
            xor         ebp,[SSE\_TABLES+1+8*esi+edi]
            movzx       esi,ah     // last row, 1
            movd        xmm0,ebp
            mov         ebp,[SSE\_TABLES+3+8*esi+edi]
            movzx       esi,ch     // last row, 11
            xor         ebp,[SSE\_TABLES+1+8*esi+edi]
            pextrw      ecx,xmm2,4 // pextrw   ecx,xmm2,4  // load 8,9
            movd        xmm4,ebp   // stop last row for now
            movzx       esi,bl     // 4
            mov         ebp,[SSE\_TABLES+8*esi+edi]
            shr         eax,16     // finish pextrw 2, 3 via movd
            movzx       esi,ch     // 9
            xor         ebp,[SSE\_TABLES+3+8*esi+edi]
            movzx       esi,dl     // 14
            xor         ebp,[SSE\_TABLES+2+8*esi+edi]
            pextrw      ebx,xmm2,3 // load 6,7
            movzx       esi,ah     // 3
            xor         ebp,[SSE\_TABLES+1+8*esi+edi]
            pextrw      edx,xmm2,6 // load 12,13
            movd        xmm1,ebp   // 2nd row done
            movzx       esi,cl     // 8
            mov         ecx,[SSE\_TABLES+8*esi+edi]
            unpcklps    xmm0,xmm1  // combine first two rows
            movzx       esi,dh     // 13
            xor         ecx,[SSE\_TABLES+3+8*esi+edi]
            movzx       esi,al     // 2
            xor         ecx,[SSE\_TABLES+2+8*esi+edi]
            movzx       esi,bh     // 7
            xor         ecx,[SSE\_TABLES+1+8*esi+edi]
            movd        xmm1,ecx   // 3rd row done
            movzx       esi,dl     // 12
            movd        xmm3,[SSE\_TABLES+8*esi+edi]
            pxor        xmm4,xmm3
            movzx       esi,bl     // 6
            movd        xmm3,[SSE\_TABLES+2+8*esi+edi]
            pxor        xmm4,xmm3
            unpcklps    xmm1,xmm4  // combine 3rd and 4th rows in xmm3
            unpcklpd    xmm0,xmm1  // combine all rows in xmm2
            pxor        xmm0,[SSE\_KEY\_OFFSET+2*16+edi]  // key addition
          // begin round 3
            ...
```

VI.6. Conclusions and recommendations for further research

In this Chapter, we presented new methods and ideas to mitigate recently demonstrated software side channel attacks. We also presented appropriate methods for discussing the effectiveness of such mitigations.

While Bernstein and OST [6, 82] also made numerous suggestions for software methods for implementations of AES that would protect against cache-based software side channels, the methods that we presented in this Chapter are different from any of the ones suggested previously. For example, although [82] suggested the possibility of using the small S-Boxes, they correctly argued that this mitigation by itself would not defeat their powerful adversaries, but would only require them to use more time. Moreover, they did not suggest to combine this with accessing all the cache lines in the small S-Boxes in every round or periodically permuting this compact S-box.

Additionally, we also introduced the concept of evaluating the mitigations relative to the power of the adversaries. This evaluation is useful in the search for efficient enough and secure enough mitigations to these new side channel vulnerabilities. Moreover, this Chapter presented specific experimental results deducted from experiments with the mitigations proposed.

We believe also that this is only the beginning of a new research path parallel to the hardware side channel research. Indeed, we are convinced that more software side channels will be discovered, which will result in new interesting mitigation methods.

Further readings

[82] *Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor,* CT-RSA, *volume 3860 of* Lecture Notes in Computer Science, *pages 1–20. Springer, 2006.*

[86] *Dan Page. Partitioned cache architecture as a side channel defense mechanism. Cryptology ePrint Archive, Report 2005/280, 2005. Available online at* `http://eprint.iacr.org/2005/280.pdf`.

[87] *David A. Patterson and John L. Hennessy.* Computer Architecture: A Quantitative Approach. *Morgan Kaufmann, 3rd edition edition, 2003.*

Part 2

SPAM analysis and prevention

CHAPTER VII

SPAM

Abstract: Nowadays spams saturate our mailboxes. Indeed three emails out of four are unsolicited. This increasing phenomenon represents a considerable resource expense for Internet Service Providers as well as for end-users. The principal defenses are various kinds of filters, but their efficiency is limited; even recent legal measures appear to merely bother spammers. The problem of spams would be tackled at its roots if spam harvest engines could not discover the user's address.
Based on a wide scope of experiments, this paper intends to provide practical numbers and figures concerning the protection of email addresses at the gathering level, analyze the evolution of the spam escalation on different sites and measure the impact of spywares.
The results enable the user to acquire a practical understanding of the spam threats, demonstrate what the actual behavior of spammers is, and increase public awareness on optimal protection of their email addresses.

Foreword: This study reveals the first concrete numbers and figures about the early stage of spam flood. Even if people have intuitions about what happens and how to protect themselves, it remain an intuition and no quantitative analysis has yet been published.

VII.1. Introduction

VII.1.1. History

VII.1.1.1. *Context.* Since the deployment of the Internet in the late 60's, the number of interconnected computers has boomed from a couple to nearly 300 million nowadays [133]. There are comparable expansions for web servers (about 57 million in December 2004) and for professional and personal accesses to the World Wide Web as well as for the global daily traffic flow.

The major part of this volume (80% according to [20]) is related to peer-to-peer (P2P) file sharing. Then come mainly the Web browsing and finally electronic mails (emails), File Transfer Protocol (FTP), Remote login (telnet), Instant Messaging (IM) and Media Distribution (audio and video streaming).

Even though emails only represent a tiny fraction of the traffic volume going around public IP networks (a little more than 1%), it involves a gigantic amount of messages: around 31 billion worldwide per day in 2002 and perspectives claim more than 60 billion for 2006 [131]. However this should be carefully considered: other sources give higher numbers (about 220 billion for 2004). Email presently stands out as the most popular communication mean ever. All those figures would not cause any problem but for spams: unsolicited commercial email (UCE), also known as unsolicited bulk email (UBE) or junk mail.

The meaning of *spam* has changed with time. At the origin, a message was called spam when widely broadcasted all over newsgroups. But later junk emails were labeled as the Hormel Foods' cans (SPiced hAM or Shoulder Pork and hAM), in reference to the Monty Python's Flying Circus sketch, where everything on a menu contained spam, wanted or not. Refer to [118] for the complete story.

Until recently, spammers formed a small community (200 members were suspected to be responsible for 80% of the worldwide spams [113]). They principally used *spiders* or *crawlers* (spambots) to harvest email addresses on web sites and *ratwares* to broadcast advertisements to the collected addresses. But nowadays, virus writers and hackers have entered the deal. They offered their experience in system intrusion and malicious code to enhance ratwares, make the spammers less traceable and defuse anti-spam protections. Among the 800 new viruses discovered each month, a large number attempts to take control of the computer in order to send spam. "Approximatively 40% of the spam is now sent from hijacked consumer systems" [112]. The most recent examples are MyDoom, Sasser and all their hybrid or mutant forms.

VII.1.1.2. *But what is the big deal with receiving spam?* Most spammers claim that the user who is not interested just has to delete or ignore them. This would be possible if only, those last years, spam had not become ubiquitous. It is now a major concern for ISPs (Internet Service Providers) and many others since an increasing share of the emails is identified as UCE. The precise number is controversial but more and more sources agree around 75% [57, 70]. In comparison, the ratio was 1.5% during mid 2002 and about 40% for mid 2003. Trends indicated a saturation above 80% for the next years [131].

This escalation has numerous sources: for example, the quasi impossibility to access online services without providing a valid email address, spiders and crawlers harvesting web sites to establish exhaustive lists of email addresses, exchanges and resales of such lists between spammers, *etc.*

VII.1.1.3. *Economic issues are also at stake.* In the U.S. for example, 37% of the working population has access to the Internet at work (86% of them with high speed access) and uses email in their everyday job [34]. In 2002, they were already spending 49 minutes average per day to manage their mail. With respect to the *spam to email* ratio, there is an important lack in productivity. Considering also the consumption of IT resources and helpdesk time, Ferris Research estimates in [46] the cost of spam to US organizations was already $8.9 billion in 2002. In [45] it is claimed that $41 billion have been spent worldwide on spam defense in 2004.

Moreover, the cost for a spammer is nearly zero: sending an email is virtually free and, thanks to ratwares, the automation level of spam spreading is high. Then the business becomes financially beneficial with an extremely low response rate: 1 over 100 000, instead of 1 over 100 for postal mail advertising [129]. However, the effective response rate is surprisingly high [19]. The spam process is then an extraordinary demonstration of lucrative cost-shifting: "the recipients are still forced to bear costs that the advertiser has avoided" [24]. Spams consumes CPU time and bandwidth of relay servers and ISPs[1]. In order to conserve their profit margin, the latter deflects the cost rises to the consumers.

VII.1.1.4. *How to counter the rising flood of spams?* Different levels of reaction can be suggested: *e.g.* laws and high profile actions, different kinds of detections by ISPs or servers, and filters on personal computers.

Recent official actions have been taken against spammers, mainly in the United States:

- CAN-SPAM Act: the *Controlling the Assault of Non-Solicited Pornography and Marketing* Act of 2003 became effective on January 1st, 2004. It "establishes requirements for those who send commercial email, spells out penalties for spammers and companies whose products are advertised in spam if they violate the law, and gives consumers the right to ask emailers to stop spamming them" [31]. It forbid anyone to harvest email addresses and to send messages to harvested addresses. Its application would also prevent any email sender form disguising or falsifying the sending address;
- Arrests and trials: Some spammers have been arrested and/or convicted: for example, on January 28th the hearings resume in the case of the AOL employee who is suspected of having sold 92 million addresses of AOL members [111];
- Judgements: in particular the controversial one awarding ISPs for $1.08 billion [37], end of 2004.

[1]In this project, we received spams of more than 100 KB.

Actually there are two major trends: *Opt-in* and *Opt-out*. The Opt-in or Option-in stipulates the user has the choice of entering a mailing list and is by default not subscribed. This is the position adopted by the European Union. The Opt-out acts conversely: the user has the right to unsubscribe the list his address is involved in. The mailing list owners have then a legal period to remove the corresponding email address. This is the position adopted by the United States. Both visions have pros and cons, even if the Opt-in seems more protective for email address owners. Nevertheless the effects on the *spam to email* ratio remained local and limited in time [71].

On a different way, spams can be fought by filtering them at the server level using different methods. Examples are given in the following:

- Intelligent filters like SpamAssassin [2] that use a genetic algorithm to detect and block spams;
- Checking the existence of an email address (and the IP of the sender) in a list of known spams using a Realtime Blackhole List (RBL);
- Preventing user's computer from becoming a mail sender server without his knowledge by blocking the port 25 (the standard SMTP port);
- Prohibiting access to *finger* servers, that provide a list of known users in a system.

Also, the same kind of actions can be taken at the end-user level:

- Using Bayes filters with the client mailer to sort spam from good emails. Nevertheless a nearly perfect segregation of spam into the *junk folder* might not be enough to stop spammers, as underlined in [68];
- Using "malicious" techniques like "Make love not spam" project launched by Lycos Europe but quickly shut down for legal and ethic problems [110].

Those actions consume vast amounts of CPU work and a noticeable percentage of the network bandwidth. User and administrator loose simplicity in email transactions since a constant management is needed to maintain protection efficiency. Moreover, spammers try to adapt spam contents, preventing them from triggering [112].

VII.1.1.5. *A new hope.* Another position of defense is to prevent the email addresses from being collected on the World Wide Web. During the 2005 Spam Conference, Matthew Prince has presented early results of the Project Honey Pot [92]. With the intention to better understand the spam-cycle, this project offers to install email addresses –on web

[2]`http://spamassassin.apache.org`

sites– that are custom-tagged to the time and IP address of the visitor: "*If one of these [email] addresses begins receiving email we not only can tell that the messages are spam, but also the exact moment when the address was harvested and the IP address that gathered it*". This information is shared and especially with law enforcement authorities in order to track down and prosecute spammers. In [16], the collect of email address is tested by the creation and the spreading of several free Hotmail addresses.

On the Internet, a wide variety of protections can be seen that are believed to mislead spammers: *e.g.* variations in the display (*nameAT-site.com* or *name at site dot com*), use of ASCII characters [51, 128], pictures [75], JavaScript [61, 115] and many others. In 2003, the Center for Democracy & Technology studied the way the spammers acquire the e-mail addresses, by using dozens of freshly created addresses [21]. But, to our knowledge, no result has been published nor is available concerning the quantitative efficiency of the possible protections. A web site could obfuscate an email address with custom functions but using such protection would probably take more bandwidth to be downloaded to the user's browser and a part of his CPU time will be spent displaying it correctly. *Is it worth the cost?* All the possible tricks present different protection and *user-friendliness* levels: for example, the pictures do not enable the user to *copy-paste* the email address and a display will not be clickable without *mailto*. Similarly, what is the influence of the position of the page on a web site? Would an email address on the front entry page be more spammed than an email address on a internal page? Moreover, common web languages provide a flexibility in the creation of hyperlink. Does the link nature matter? And are all types of link resolved and followed by spam spiders?

This contribution tries to fill the gap on this topic and intends to give proven and quantitative answers concerning the efficiency of the protection against spam, at the gathering level.

VII.1.2. Project Overview

This Chapter follows the structure of the project, where three phases were set up. The first one focuses on a passive user behavior, as if all received emails were read offline. Web sites are created with various domain names while each page contains a set of email addresses with different levels of protection. Those pages and sites constitute honey pots and all emails sent to the domain addresses are received and stored for further analysis. The notoriety of a site is tested as well. Section VII.2 details the structures of this phase while, in Section VII.3, main conclusions are drawn after 3 months. Some hypotheses are also proposed, especially concerning the philosophy of the email address's gathering.

Phase 2 permits testing those hypotheses with a set of new experimentations. Therefore Section VII.4 places the emphasis on active profiles by modifying particular sites' properties, while other sites are kept as in the first phase for comparison matters. Other behaviors are tested as well: for example, the potential influence of posting in newsgroups or using concrete applications involving validated addresses. Section VII.5 hands out conclusions at this level.

The third phase is described in Section VII.6, wherein other influences are measured, particularly concerning the spywares that might infest a computer. Also, some addresses from the phase 1 are removed and replaced by new ones. Conclusions are given in Section VII.7.

Eventually, Section VII.8 gives the global conclusion of this study, a seven-month long snapshot of spam activity on real systems.

VII.2. Phase 1: System Description

For the sake of simplicity, the expression "e-address" will stand for "email address" in the rest of this paper.

This Section describes the conception of the trap. It goes all the way from the e-address elaboration up to the design of web sites. The system provides a lot of e-address types for testing spammers' reaction. Of course, content and structure of a site must be carefully analyzed.

In this context, the purpose of page elaboration is twofold. At first, we deal with the insertion of e-addresses in all possible locations of the page code. Meanwhile, the content should look real and meaningful. There is then a divergence due to the fact that e-addresses should be clearly visible in the code for spam engines, without chocking the human reader. Another interesting point is to avoid the use of OCR[3] softwares in the spammers processing stage as we want to minimize this method of harvesting. Moreover, the links between pages must be as different as possible to study the spam spiders' navigation.

This paper also focuses on the notoriety and age of a site. It is commonly admitted that these two factors would influence its referencing and, therefore, its leaning towards spam.

At the end, the physical conception is briefly described.

[3]Optical Character Recognition: recognition of characters in a pixelated graphic file and conversion of them into a regular text file, see `http://en.wikipedia.org/wiki/Optical_character_recognition`, rev. 5/11/05.

VII.2.1. Email Address Types

This Section describes specific HTML structures that contain explicit or implicit e-address information. It begins with basic solutions to insert a correct e-address with or without visual result in the browser. Then a clickable property is provided to enhance the previous solutions. At the end, we play with the e-address itself along with other languages used in nowadays web sites. The goal is to study which particular HTML forms trigger the spammers' reaction.

These HTML structures refer to the basic code which could be combined to build more complex ones. The number in parenthesis refers to Table 1, next to each descriptions, provides information about the HTML and its visual result in the page.

`HTML Code`	Visual Result in the Page

HTML language gives numerous ways to display/provide an e-address. The first solution is a direct display:

- With a text in the page (#4);
- With a picture containing the e-address (#36).

`name@site`	name@site
`<img src="email.gif">`	name@site

Then, ASCII can be used to encode some characters and one can insert useless HTML tags in the structure (#14 and 16).

`name&#64site&#46com`	name@site.com
`name@s<font size="1">ite.c</font>om`	name@site.com

Finally, the e-address can be enclosed to the document with no visual result in the HTML page, as a commentary or as a META tag in the header (#1 to 3).

`<!- - name@site - ->`	
`<META name=‘‘generator" content=‘‘name@site">`	

A second solution is to provide a visual object with a link to the e-address:

- In a standard form (#10);
- In a standard form without “mailto” (#7);
- In a standard form with a link different than the displayed e-address (#8 and 9).

`<a href="mailto:name@site">name@site</a>`	name@site
`<a href="name@site">name@site</a>`	name@site
`<a href="mailto:other@site">name@site</a>`	name@site

In the previous examples, changes occur only in the HTML code. Now, let us play with the e-address form and change it to an estranged one. In this case, a manipulation is needed to retrieve the real address. The corruption is left to the user's imagination. Here is a short list of possible tricks:

- Adding a specific text describing the manipulation to reveal the e-address (#11 and 12);
- Changing "@" and/or "." with another explicit or implicit symbol (#13, 21, 26 and 31).

`nameRwithoutR@site`	nameRwithoutR@site
`nameATsite`	nameATsite

Eventually, HTML language can call other interpreters like *JavaScript*, *Java* or *Flash* (#37 and 38). All these objects can be used to display an e-address and sometimes make it clickable.

As an example: in the case of *Javascript* (#20), a function can be used to muddle the e-address in the code. When it is interpreted, it displays the e-address in the browser. If spammers want to collect this sort of information, they must execute the *JavaScript* codes on a page. Such operations require substantial of processing power.

In last example, spammers can use OCR software to analyze the web page and reduce the processing phase. The design of our page prevent as much as possible this sort of interaction (as described in Section VII.2.2).

In conclusion, each page in our web sites has 38 different types of e-addresses, each of them resulting from a *mixture* of some of the basic characteristics described above. Table 1 provides all e-addresses. For information, each e-address has a unique structure to allow easy classification and statistics but this paper did not shows the real form to prevent the system from any perturbation.

Description	**Code Example**
1. META tag	<meta name="email" content="add@site">
2. Comment in head	<head><!- - add@site - -></head>
3. Comment in body	<body><!- - add@site - -></body>

Continued on next page

Description	**Code Example**
4. Std. e-add.	add@site
5&6. Std. e-add. with link to other e-add.	<a href="other@site">add@site</a>
7. Std. e-add. with link to the same e-add.	<a href="add@site">add@site</a>
8&9. Std. e-add. with link mailto to other e-add.	<a href="mailto:other@site">add@site</a>
10. Std. e-add. with link mailto	<a href="mailto:add@site">add@site</a>
11. Tricky e-add. "without"	add10without10@site
12. Tricky e-add. "without" with link mailto	<a href="mailto:add10without10@site"> add10without10@site</a>
13. Tricky e-add. with spaces	add AT site com
14. Tricky e-add. ASCII	add&#64site
15. Tricky e-add. ASCII with link mailto	<a href="mailto:add&#64site">add&#64site</a>
16. E-add. with useless HTML tag	add@h<font body color="#FFFFFF" size="1">sit</font>e
17. E-add. with Javascript (1)	<script>document.write('add\x40site')</script>
18. E-add. with Javascript (2)	<script>document.write('add@'+'site')</script>
19. E-add. with Javascript (3)	<script> document.write('add___@__site'.replace (/_/g,''))</script>
20. E-add. with Javascript (4)	Fct that transform number to an e-add., visible in the browser
21. Tricky e-add. AT	addATsite
22. Tricky e-add. AT with link	<a href="addATsite">addATsite</a>
23. Tricky e-add. AT with link mailto	<a href="mailto:addATsite">addATsite</a>
24. Tricky e-add. AT with real link	<a href="add@site">addATsite</a>
25. Tricky e-add. AT with real link mailto	<a href="mailto:add@site">addATsite</a>
26. Tricky e-add. H	addHsite
27. Tricky e-add. H with link	<a href="addHsite">addHsite</a>
28. Tricky e-add. H with link mailto	<a href="mailto:addHsite">addHsite</a>
29. Tricky e-add. H with real link	<a href="add@site">addHsite</a>
30. Tricky e-add. H with real link mailto	<a href="mailto:add@site">addHsite</a>
31. Tricky e-add. Random @	add(random)site
32. Tricky e-add. Random @ with link	<a href="add(random)site">add(random)site</a>
33. Tricky e-add. Random @ with link mailto	<a href="mailto:add(random)site">add(random)site </a>
34. Tricky e-add. Random @ with real link	<a href="add@site">add(random)site</a>

Continued on next page

Description	Code Example
35. Tricky e-add. Rnd @ with real link mailto	<a href="mailto:add@site">add(random)site</a>
36. GIF e-add.	<img src="e-add.gif'>
37. Flash e-add.	Flash code
38. Flash e-add. clickable	Flash code with link mailto in it

Table 1: List of the 38 e-address' types used in our project.

VII.2.2. Page Structure

In this context, the purpose of the pages is twofold. The effective reason is to deal with the insertion of e-addresses in all possible locations of the page's architecture, while its content should look real and meaningful. There is then a divergence due to the fact the e-addresses should be clearly visible in the code for spam engines, without chocking the human reader.

In this Section, the structure of the pages is briefly described and particularly the insertion of e-addresses in them.

A page is basically divided into two blocks. The first part is the *header* and contains various HTML tags. A first e-address is inserted in clear, in the content of a META tag. A second one is inserted as an HTML comment. Although present in the source code, none of these e-addresses are visibly displayed in the Internet browser.

The *body* part contains the browser-visible content of the page. This one varies depending on the position of the page within the sites. After the content, an HTML comment includes another e-address. Then comes a gray-shaded text (*No use of these:*), preceding all remaining e-addresses. They are all formatted in size 1 and written in white on a white ground, in order to be kept as discrete as possible and to prevent the use of OCR software. E-addresses represented by pictures or flash links are gray-shaded as it would be a non-sense to create a white text in a white background in this case. We will then take into account the use of OCR techniques.

Additional features can be required to display the page correctly, as `script` tags to insert *JavaScript* functions at the end of the header. These additional features are tuned by parameters in the generation program.

The file size of the pages is about 7 to 14 kbytes, the *Flash* files about 3 kbytes and the pictures less than 1 kbyte; in order to permit quick displays.

VII.2.3. Site Structure

In this Section, the generic structure of the sites is depicted and afterwards are presented the differences between the resulting sites in term of notoriety and age.

The project sites are composed of various pages, located in different directories and linked together by sundry hyperlink varieties. The aim of such connections is to evaluate the capabilities of the spam engines: whether they are able to follow links or not, which kind of links, what depth from a given page, *etc.*

For the creation of one site, ten pages are generated (refer to Figure 1).

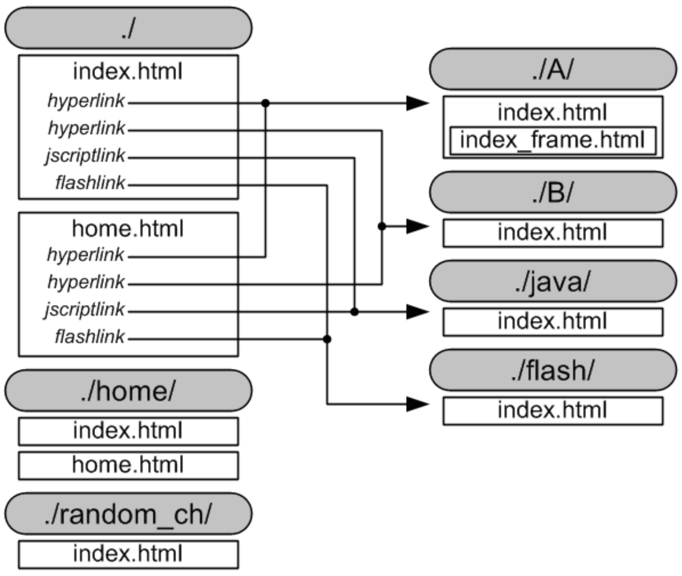


Figure 1: Structure of a site – Each rectangular box represents a web page, with a set of e-addresses. The gray rounded box contains the position in the arborescence.

The web server is configured to present *index.html* as default homepage. In the same directory, there are *home.html* and the required files (pictures and *Flash* files). We did not provide a link pointing to *home.html*. Both pages look alike at first sight, but for the e-addresses.

From those pages, four hyperlinks are available. The first two go to different directories and refer to an internal *index.html* file. The third one is a *JavaScript* link targeting *./java/index.html*, while the last one is a *Flash* link to *./flash/index.html*.

The *./A/index.html* page is composed of a frame containing the classical set of e-addresses but they are not visible at all in frame-capable browser:

this is the “no-frame” page. Its single frame is *./A/index_frame.html*, which also contains its own e-addresses. This is done in order to test the ability of spam engines to resolve this type of structure and access *./A/index_frame.html*.

Finally, three other pages are available:

- *./home/index.html*,
- *./home/home.html* and
- *./random_ch/index.html*).

Even though those pages are available, we did not create any hyperlink to reach them. The *./home/* directory has been chosen because it is believed guessable, while the other folder's name is formed by eight random characters and would not be reached unless through an exhaustive search.

One site does not comply with this structures; this will be described in the next Section.

VII.2.4. Site Type

The notion of *promotion* is introduced here as ways for a site to get a better ranking in search engines (as for example Google or Yahoo). It can be done through a dedicated software, for example, which helps the webmaster to optimize the keywords in the *meta* tags. Promotion affects the notoriety of the site. Frequently visited and well referenced sites clearly have a better notoriety than brand new domain names. The project sites possess different characteristics: old or new domain name, with or without promotion. For this study, five web sites have been created (refer to Figure 2):

- A site on a new domain name, with no promotion (site1). This is the reference site as we presume that no spam will hit this domain;
- A site on an existing but unused domain name, without promotion (site2): “*What is the influence of a site's existing time? And is the domain name playing a role in the detection by spammers?*”;
- A well known Partner site, already promoted (site3). The e-addresses spread over this site are not related with the hosting domain but linked to site2. Their form (“name@web.site2”) makes them distinguishable): “*What are the influences of the age and the notoriety upon spams? Two sets of e-addresses sharing the domain name will influence each other's spam infection?*”;

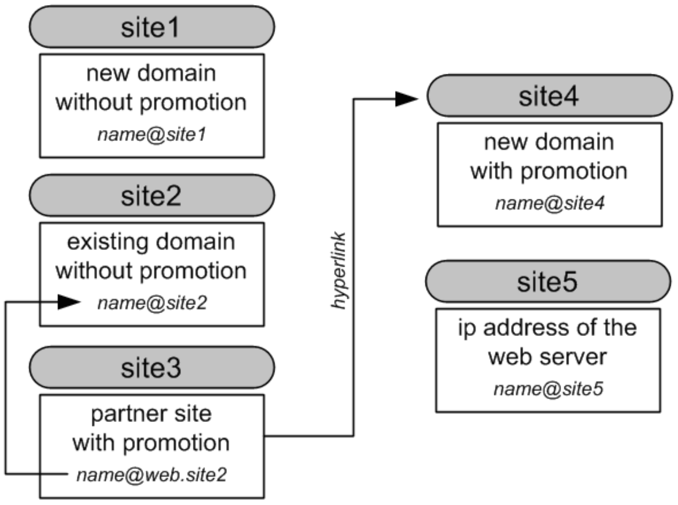

site1
new domain without promotion
name@site1
site2
existing domain without promotion
name@site2
site3
partner site with promotion
name@web.site2
hyperlink
site4
new domain with promotion
name@site4
site5
ip address of the web server
name@site5

Figure 2: Sites properties and relations between them.

- A site on a new domain name, with promotion (site4), with a hyperlink from site3: "*What is the benefit of the promotion in the case of new domain names? Does a hyperlink from an often visited site have any consequence?*";
- A site accessible by the *IP address* of the computer hosting the web server (site5): "*Do the spammers search engines try and reach the web server by IP?*"

For the Partner site (site3), only two pages are involved: the homepage and an internal page reached directly by a homepage's hyperlink. In fact, as each pages have 38 e-addresses, we will only have 76 e-addresses for this site against 380 for the others, but 2 pages are already relevant in this study. We will take into account this configuration in our results.

VII.2.5. Physical System

The previous parts describe the philosophy of the honeypot:

- 38 different e-addresses in one page;
- 10 pages per site (2 in the case of the Partner site);
- 5 types of site.

Only one computer is needed as a web and mail server. Pages are static and generated once and for all with an external program for efficiency matters.

The tag structure is respected to allow browser compatibility and e-addresses are spread throughout the code.

Several tools monitor the web sites/mail activities and keep logs for further treatments.

Also, we need to access and change popularity of web sites used in this project. One of them, with already a good popularity, was generously provided by a Partner[4]. The others were recently bought and they became more or less available in search engines with the help of a ranking software. This situation does not interfere with the model as only one popular site is needed in this study.

VII.3. Phase 1: Results

Table 2 provides a representation of the quantity of spams sent per month and per site from September 18th, 2004 to April 19th, 2005. Phase one focuses on the first three months. During this period of time, the system received 1872 spams (1733 to project's domain and 139 directly to the mail server as described in Section VII.3.1).

	Phase 1			*Phases 2 and 3*				
Month	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Total**
Site1	0 (0)	0 (4)	0 (0)	0 (3)	0 (10)	0 (12)	0 (11)	40
Site2	1 (14)	0 (22)	500 (24)	406 (41)	685 (33)	912 (43)	793 (82)	3 556
Site3	297 (0)	289 (0)	570 (0)	656 (0)	1 369 (2)	2 140 (0)	2 280 (36)	7 639
Site4	1 (0)	0 (2)	9 (0)	161 (1)	138 (6)	426 (3)	433 (9)	1 189
Site5	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0
Mail Server	6	23	110	93	54	171	93	550
Total	319	340	1 213	1 361	2 297	3 707	3 737	12 974

Table 2: Quantity of spams per month and per site – Mail server refers to spams directly sent to the mail server without any domain in the e-address. The first number refers to spams sent to one of the e-addresses listed in the web site, second number in parenthesis refers to spams received to an e-address not present in the web site.

First spam was received September 18th to a standard e-address from the homepage of the Partner site.

VII.3.1. Domain Structure and Hits Influence

This Section begins with some results of the evolution of each domain and its corresponding quantity of spam.

First, no spam came from site5 hosted directly by the web server (reachable by IP). The reference site1 (not promoted on the Internet) received

[4]The Google PageRank is 0 for site1, site2 and site5; 7 for site3 (the Partner site) and 5 for site4.

4 spams, all of them to the "webmaster" user (see Table 2). It confirms that spammers have no interest of contacting directly the IP of web servers and that they use at least common techniques to discover new web sites, such as search engines or direct links from other sites.

Table 3 is the list of *page hits* for each web site involved in this phase. As expected, Partner site3 has a lot of visitors. Site4 also has a lot of hits. This must come from the promotion in search engine by a specialized software and through site3 hyperlink. The other sites seem to remain in the background. At this time, we expected spam to come mainly from Partner site3 and site4 in majority but the results do not really confirm this intuition.

Web Site	**Hits**
Site1 (reference site)	43
Site2 (domain from site3, no promotion)	109
Site3 (Partner site)	775 798
Site4 (link from site3 and promotion)	6 884
Site5 (IP from web server)	22

Table 3: Number of hits by web site.

At the beginning, we believed that a direct link from a heavy hit web site could be an excellent mean for spammers to find a new domain and its corresponding e-addresses. It is actually true but the flooding is not as important as in the case of site2.

Figure 3 shows the percentage of spams per domain balanced by the number of e-addresses in each site as Partner site3 had less e-adresses than the others (refer to Section VII.2.4).

We can see that 79% of spams come from Partner site3. An old web site with good ranking in search engine is clearly an excellent choice for spammers. Then only 2% come from site4 despite the number of visitors and promotions. Surprisingly, site2 came second with 15% of spams. This site has no hit, no hyperlink from other web sites and no existence in search engine. More precisely, the only link between site2 and the Partner site3 is the domain of e-addresses: `name@web.site2` for Partner site3 and `name@site2`. In fact, site2 only has only one specific configuration common with the Partner site, the domain of e-addresses:

site2	name@site2
Partner site3	name@web.site2

Domain name site2 has another characteristic: we acquired it one year before the project began, but this specification did not interfere with our model. In fact, this domain was not receiving any spam that we were aware of before this study. Spammers seem to use domains inside e-addresses to find their target.

Finally, another unexpected result comes from Figure 3: an important number of spams is sent directly to the mail server without any domain specified in the address. If this sort of email is sent to a server, the latter tries to find the local corresponding user in its database. In our case, 4% of spams were sent to “postmaster”, “root”, “webmaster” or “info” user.

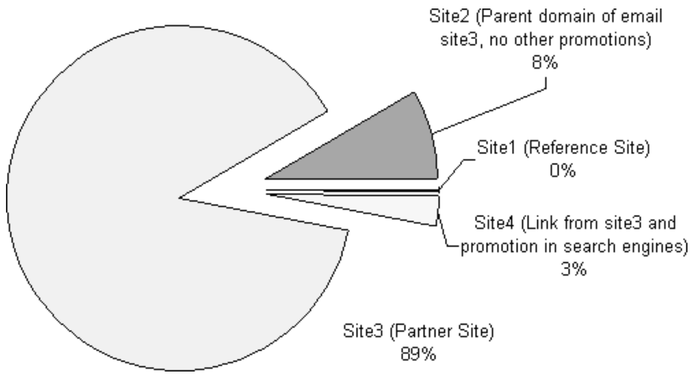


Figure 3: Percentage of spams per domain (balanced by the number of e-addresses in each site) – Site5 is not displayed as it is irrelevant in this case.

VII.3.2. Page Position and Intersite Link Influence

Recall the design of all web pages involved in this project (as detailed in Section VII.2.3). Only specific ones have been visited by spammers to retrieve e-address information.

Table 4 shows the quantity of spams by type of page. The quantity of spams per page from Partner site3 must be discarded as it is irrelevant in this context. We decide to show them for information only.

In phase 1, no page was more triggered than another. This conclusion changes in next phases with more data (cfr. Section VII.5.1).

Flash and hidden pages do not appear in Table 4, as exhaustive search and complex links seem to be of no interest for spammers. It would be a waste of time and power for them to resolve these types of links, for the moment.

VII.3.3. Email Address Type Influence

This project provides 38 types of e-address per page as described in the Appendix. Figure 4 depicts the results for phase 1. The system

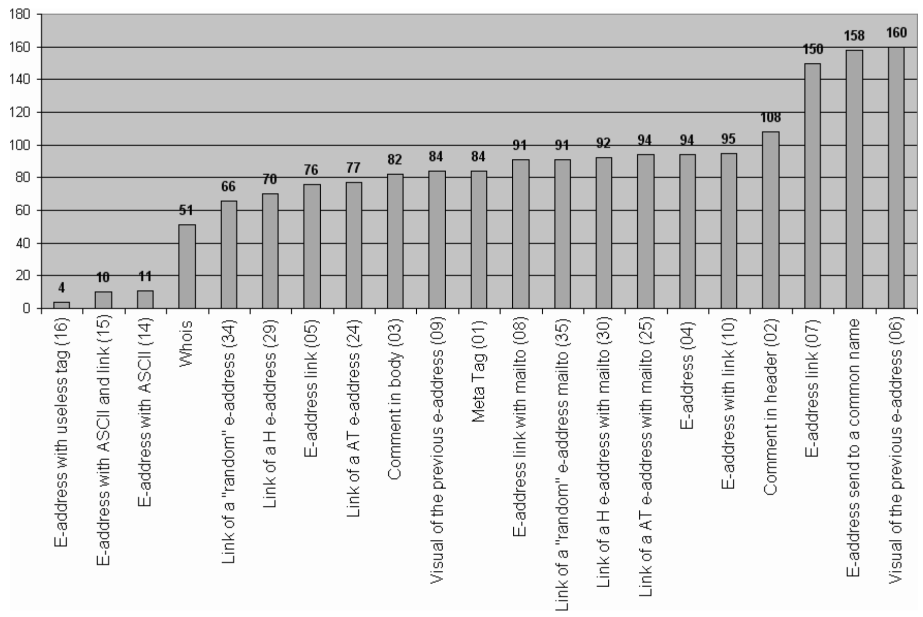


Figure 4: Quantity of spam by e-address type for phase 1 – For each type, numbers in parenthesis refer to numbers in the Appendix. E-addresses not reached by spammers are not displayed in this figure.

Page	**Site2**	**Site3**	**Site4**	**Total**
./index	85	677	8	770
./B/index	89	351	0	440
./A/index	87	(-)	0	87
./A/index_frame	95	(-)	0	95
./java/index	91	(-)	0	91
Total	447	1 028	8	1 483

Table 4: Quantity of spam per page position in phase 1 – Site1 and site5 did not receive any spam to e-addresses listed in web pages. (-) denotes there is no corresponding page for the site.

received spams to incorrect e-addresses of type 11 and 12 (spammers did not remove “XwithoutX” tag): They do not appear in Figure 4 as they must not be taken as spams reaching the user mailbox.

The conclusion can be divided into three parts:

- All e-addresses clearly written in the page code have more or less the same spam activity/sensibility. Comment, presence or not of the “mailto”, e-addresses directly provided on the page

or as a link; all these characteristics do not influence spammers activity.
- Only two protections fail to prevent e-address from being harvested: using ASCII instead of character (type 14 and 15) and inserting useless HTML tags in the e-address (type 16). The quantity of spams received in those cases is small (1%), but these are the only ineffective protections.
- E-addresses provided during registration of domain names which can be found in the Whois[5] database also trigger spammers attention (2%).

Obviously, webmasters can protect e-addresses from being harvested by using a simple protection as changing "@" by "AT". Choosing a more complicated solution could protect these more effectively in the future. It remains clear that providing e-addresses with a simple form must be banished. Some studies claim that using ASCII codes protects the user from spammers. Our results question this allegation. Therefore, we encourage webmasters to use another system.

VII.3.4. Conclusion

With these preliminary results, webmasters can efficiently prevent spammers from gathering e-address information from their sites:

- Providing e-addresses in an internal page referred by a complex link;
- Using a simple but efficient technique to obfuscate e-address from the page code;
- Using the domain of the page for the e-address domain or at least a unique domain to prevent spammers from finding other sites to explore.

VII.4. Phase 2: Disturbing the System

The first phase of this experiment gives an idea of the impact concerning spam flooding with a passive behavior. The user has some e-addresses spread over the Internet and he never does anything to prove the validity of his e-address (for example, he has a perfectly trained spam filter, therefore even if he receives spams, he is never confronted with it nor has he ever got to react to them).

After 3 months, a second phase is launched and several purposes are targeted. The first is to experiment an active behavior, where the user discloses the validity of his e-addresses by different possible ways:

[5]Whois is a query - response protocol to search a database of domain owners.

- Some spams happen to be written in HTML and contain links to pictures, which are not sent with the email but reachable through the Internet. A closer look could reveal a strange code: *e.g. SRC="http://.../jpg/928567366"*. A large amount of mailers used to resolve HTML links when one reads the email in order to display it correctly. Then the web site receives a request to the picture *928567366*. Most probably, this link is virtual and has been bonded to the particular e-address the spam was sent to. Therefore, a simple skim of the email may command the resolution of the link and therefore reveal that this e-address is valid. For the spammers, this raises the value of the e-address. According to [83], eight spams out of ten contained tracking codes.
- Forums and newsgroups are also places where people are using valid e-addresses (sender e-address, reply-to e-address or e-addresses in the body).
- Electronic encryption and certification tools often provide online directories of identities (that may include e-addresses). For example, PGP detains open-access servers storing public keys for users' e-addresses. Moreover, PGP intends to generate a catalog of validated e-addresses[6]. The user can upload his keys and then follow a link in a sent email. By this way, PGP signs its keys that is consequently validated by that trusted third-party. No visual protection is set up in order to prevent automatic harvesting.

All those examples show ways to unveil valid e-addresses. This second phase shows the quantification of the incurred risks of such behaviors. It is worth noticing that most of the time there is no way to avoid it: *e.g.* in PGP, the user has to specify a real e-address, for encryption as well as signature, otherwise, the system is useless.

The second motivation for this phase is to verify the hypothesis brought by the results of the first phase. In Section VII.2.4, the use of the same domain name for different kinds of address has been introduced (name@web.site2 and name@site2). The results brought to light that the non-promoted new site (site2) has been more spammed than the promoted one (site4). Among other explanations, this could be due to the *proximity* of the e-address domain. This second phase intends to validate or not this assertion, as well.

The following actions have been taken:

- Three domains were unmodified in order to converse clean references (site2, site4 and site5).

[6] `https://keyserver.pgp.com/`

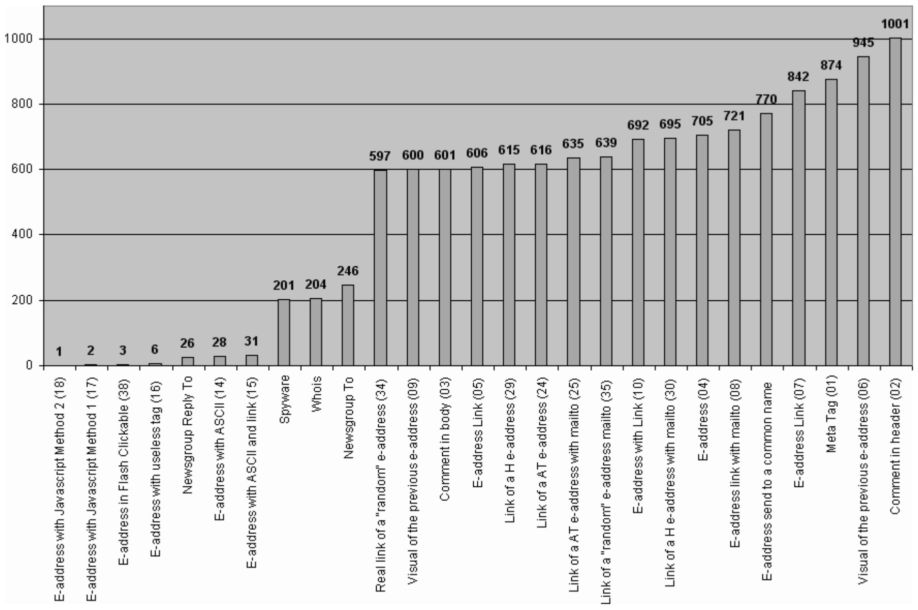


Figure 5: Quantity of spam by e-address type for all phases – For each type, numbers in parenthesis refer to numbers in the Appendix. E-addresses not reached by spammers are not displayed in this figure.

- The most *spam-polluted* domain (site3) has been taken to test the link resolution in the emails. About all of the 900 emails were read with a mail tool (which by default resolves the picture links). About 10 % of them had such links. It is the only modification on this domain.
- A new domain name has been bought (site6) and was used to create a set of new e-addresses. They have been used in newsgroups: a rapid first time in popular ones, then in newsgroups which seemed to be more polemical or more focused on what spammers offer. A corresponding web site has been created with unprotected e-addresses.
- A couple of PGP keys have been created with e-addresses from site1. The keys have then been sent to the servers and to the Verified Key Service.

VII.5. Phase 2: Results

After five months (phase 1 included), the system has received 5530 spams as shown in Table 2.

VII.5.1. More Data for Phase 1

Firstly, the results of phase 1 can be enhanced with more data. Site1 and site5 remain in the same condition: site1 received spams to common names and site5 none.

Page	**Site2**	**Site3**	**Site4**	**Total**
`./index`	603	3 339	762	4 704
`./B/index`	651	3 821	0	4 472
`./A/index`	696	(-)	52	748
`./A/index_frame`	593	(-)	364	957
`./java/index`	574	(-)	0	574
Total	3 117	7 160	1 178	11 455

Table 5: Quantity of spam by page position in all phases – Site1 and site5 do not receive any spam to e-addresses listed in web pages. (-) denotes there is no corresponding page for the site.

We remind that numbers for Partner site3 are only for information in Table 5 and are not taken into account in this subsection. In phase 1, no conclusion can be drawn for page position. Now, we have more attractive results: the homepage seems to be more visited (32%). The last place is given to a page linked by a javascript hyperlink (13%). This result is not surprising.

In the case of a frame page, we expected that if spammers would use recent softwares to collect information, they may easily find e-addresses from the two pages involved in a frame structure. At least, they may consider the content of the frame as a normal one, with an hyperlink from the homepage. Spiders had frame capacities and they missed e-addresses from the "no-frame" page. In fact, they had the same behavior for a frame page as for a classic one (15%). Webmasters could not used this properties as e-addresses in the "no-frame" page will not appear in a frame capable browser.

In conclusion, spammers react to special links but they do not follow links using flash or other sophisticated languages.

The most important difference with phase 1 appeared in the e-address protections. After six months, 3 new types of e-address were targeted by spammers, all of them received only one spam (Figure 5). These new types are: e-addresses with flash protection (type 38) and with javascript protections (types 17 and 18):

- Javascript protections 17 and 18 are pretty simple: e-addresses from the page code are obfuscated but remain in clear in the browser. These protections only use basic javascript functions

like "+" or ASCII representation. More complex ones like type 19 or type 20 use "replace" function or even more complex ones. We tend to presume that recent spiders choose to compute simple javascript code and that OCR techniques were not involved at this time.
- The break of the flash protection is more surprising. At first, we can think to OCR techniques but this induces that image protections (type 36 and 37) may be broken as well. This is not the case and the only difference is the "clickable" property of this flash code. We will then assume that spiders can trigger a "click" on advance languages. Nevertheless, we can never be certain that an e-address has been acquired by a spider rather than manually.

In conclusion, webmasters should restrain the users from using the homepage to display e-addresses and they should use complex javascript protections or images without "clickable" property.

VII.5.2. Active User behind Spam

With some types or configurations, email clients can go on the Internet and display HTML email as a web page. Some spammers use this behavior to validate e-addresses. Another technique, less sophisticated, is to ask the user's client to confirm the reading of the junk email. At the beginning of phase 2, all spams for site3 have been opened with this type of client. We found:

- For the homepage, 41 spams with web images but only 3 with an explicit reference to validated e-addresses.
- For the internal page, 61 spams with web images and 32 with an explicit reference to validated e-addresses.

As seen in Figure 6, there is a significant increase over time for the internal page. This inflection in the curve appears one month after we let the e-addresses been validated. However, the system did not change and hits from both pages remained the same. As a matter of fact, every e-address receiving spams in the internal page was targeted by a validating spam. This is not the case for the homepage which had only 3 e-addresses confirmed.

Validation with web images seems to induce more spams, however few spams have these check techniques as a lot of clients are now protected, especially the most popular ones.

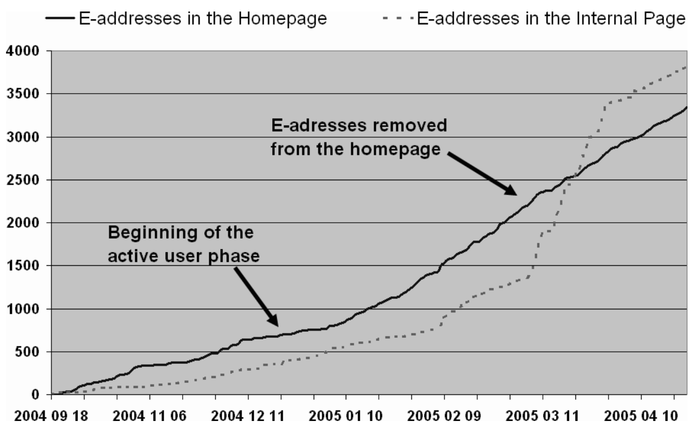


Figure 6: Evolution of spam for the homepage and the internal page of the Partner site3.

VII.5.3. Emails in Newsgroup and new Locations

PGP directories did not attract spammers (no spam received) but newsgroups seem to win the challenge: the mail server collected a spam just after the first message to a newsgroup was sent.

Spams received from newsgroups after twenty days of activity already take 2% of all spams received in four months. Newsgroup harvesting seems to be a technique actively used by spammers (Figure 7).

Eight newsgroups were chosen, each of them classified by activity (*i.e.* the quantity of mails sent to them in a short period of time). We expected to receive more spams from dynamic newsgroups but, as a matter of fact, the results did not confirm this hypothesis. 60% of emails came from little newsgroups, with a very specific subject. Spammers use newsgroups category to target their audience. Internet users should not provide their e-addresses in any newsgroup, even small ones.

An interesting remark can be made concerning the spammer's harvesting system. They collect e-addresses from the emails header (90% from "To" and 10% from "Reply-To" fields) but discard information from other places like body part. It seems that, if one wants to prospect all existing newsgroups, retrieving entire emails consumes a huge amount of bandwidth compared to retrieving only email header. Currently, users can provide their real e-address in the body of a message but this situation may change quickly if spammers adapt spiders.

Moreover, e-addresses on site6 received no spam at all. This is probably because the gathering by newsgroup is efficient, as it is; information gained by further treatments would have marginal values.

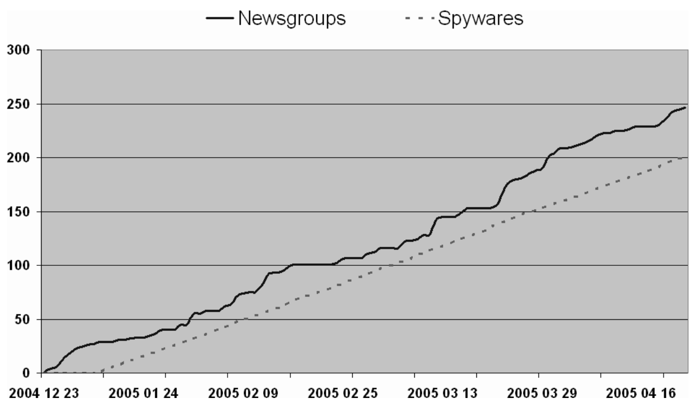


Figure 7: Quantity of spam for newsgroup and spyware traps – Spyware results are discuss in Section VII.7.1.

VII.6. Phase 3: Impact of Spyware and Variation of e-Addresses

The third phase principally targets the influence of spywares towards spam. Spyware is a program collecting personal information on a user's computer and sending it back to the spyware's authors. It can then be seen as a combination of the philosophies of browser cookies (tracking surf data) and viruses (malicious behaviors), with financial targets. In most cases, they are silently deployed during Internet navigation or during the installation of legitimate programs. Programmers may legally incorporate those spywares in their installations, as long as the user somehow agrees. This is usually insidiously done in the "Licence Agreement", since the widest majority of users never bother to read them. This way, spyware companies can sponsor freeware versions of some popular softwares.

Besides the aspects of privacy, unprotected computers are easy targets for spywares and some symptoms can betray their presence: lack of performances (CPU time, memory, bandwidth, *etc.*), instability of the operating system (crashes, hangings), connection troubles, *etc.* Some programs even attempt to control the phone modem and dial expensive numbers.

Spyware escalation opens a new business opportunity for companies offering anti-spyware protection. The size of this market is comparable to anti-virus and anti-spam ones. It is then not surprising that giant software or operating system developers jumped in the bandwagon.

Among the information sought by spywares in the computer, there are logins, passwords, web preferences or habits and also e-addresses. A considerable proportion of spams is believed to be sent to e-addresses gathered by spywares [112]. To our knowledge, no precise number concerning the impact of spywares on spam is publicly available.

This third phase has been set up to quantify this influence and improve previous results:

- A freshly installed Windows XP Service Pack 2 computer was chosen as a martyr (for the sake of science). No anti-spyware or anti-virus software nor firewall had been deployed (the default Windows firewall has been disabled). This computer was continuously connected to the Internet and a couple of hours were spent surfing the Internet, visiting all types of sites. Forms have been filled with new e-addresses without any subscription to newsletter, if the choice was available. Some programs have been installed, as they were known to install spywares in addition of their main feature.
- The impact of e-address variation and disappearance was tested as well. All e-addresses from the homepage of site3 have been removed.
- Another change occurred in the system but it was not foreseen. The domain name server changed its policies concerning the *Whois* database: it enabled a visual protection of the contact details of all the consumers, including the e-address. Previously, the consumer had the choice to hide his e-address or not.

This phase started five months after the begining of the project, although all actions were not directly applied.

VII.7. Phase 3: Results

VII.7.1. Spyware Impact

The test computer contained more than 50 instances of spyware (for example, the Microsoft AntiSpyware application finds 65). The system became unstable and had permanently high CPU utilization. The only applications running on the system were Outlook Express, which had 4 e-addresses in its e-address book, and MSN Messenger which was running with a few contacts.

We began to receive spam on this computer immediately after the 3rd party spyware-infested programs were installed. The target e-address was used in a web site when installing one of the spyware applications. We denied the registration of this e-address in mailing list and we declined the company the right to exchange it with other "commercial partners". After one day, despite all previous precautions, the company appeared to have already shared it to (other) spammers.

As a first conclusion, spyware companies deliberately use and spread the e-address given during the installation against our will.

In Figure 7, the amount of spam coming from spyware programs seems very regular. In fact, a lot of them come from the same company. We tried to disable our subscription several times with no success and spam from other advertisers arrived only occasionally.

One e-address from the outlook book received 7 spam messages. The MSN Messenger addresses were not affected.

Our second conclusion is at least one spyware violated the user's privacy by scanning the computer and sending an e-address back to its company. We only observed this effect but it might have sought and sent much more sensitive data.

VII.7.2. Modification of the Whois Database

During the registration of domain names for this project, we decided to use two specific e-addresses. The first one is visible when one queries the whois database. The other is hidden by the domain name server (a "strange" e-address is provided to the whois inquirer instead of the real one). At the beginning, the mail server started to receive spams to the unprotected whois e-address (181 spams at this time) but none was sent to the protected one.

When the domain name server changed its policies and decided to protect all whois e-addresses by a captcha[7] protection, this had a surprising bad effect. The quantity of spam currently received to the visible e-address remained the same but the other previously hidden began to receive spam too (see Figure 8).

Whois database is a good place to harvest valid e-addresses. Spammers seem to use enhanced spiders to look into it. Although OCR techniques could help in this case, we suspect human actions behind the harvesting. It would represent a huge work, but a really profitable result since all e-addresses are (supposed to be) valid. Domain name servers must try to use stronger solutions to protect their database. Three months after implementing this new protection, our domain name server turned it off and reverted back to the old one.

[7]Completely Automated Public Turing test to tell Computers and Humans Apart: challenge-response test used in computing to determine whether or not the user is human. `http://en.wikipedia.org/wiki/Captcha`, rev. 5/10/05.

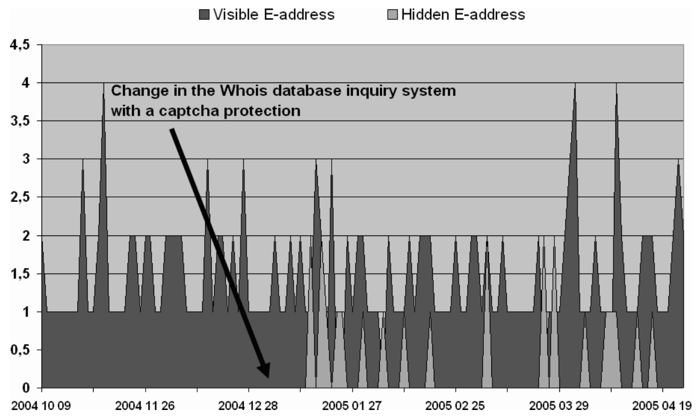


Figure 8: Quantity of spam by type of whois e-address.

VII.7.3. E-addresses removal

It is generally accepted that removing e-addresses from web page has a significant impact on spams: the quantity may rapidly decrease. However, this conclusion does not confirm the usual spammers philosophy: using all possible e-addresses.

Six months after the beginning of the project, we decided to remove all e-addresses from the homepage Partner site3. We intended to curve down the quantity of spams received specifically to these e-adresses and perhaps for the whole domain. Despite this heavy modification, no evolution could be seen at all (Figure 6): the quantity of spams grew as usual.

As we presume about the spammers philosophy, when they find an e-address, they use it as much as possible without checking its real existence.

VII.8. Conclusion

In conclusion, Figure 9 describes the quantity of spams received per day by one of the most active e-addresses.

The project described in this paper targets the quantization of the early stage of the spam-cycle. Through its three phases, a wide variety of traps has been set up to analyze the effectiveness of e-addresses protections, the actual behaviors of spam spiders and the danger of communicating valid e-addresses. In this framework, although it is impossible to distinguish the gathering by spider or by human, spammers would rather highly automate the harvesting.

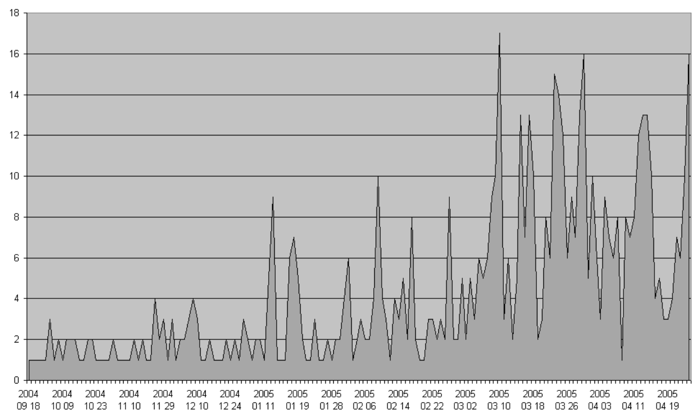


Figure 9: Quantity of spam *vs* time (day) for an active e-address – This standard e-address (type 6) is located on the Partner site, in an internal page.

These honey pots brought to light some concrete spammers' actions about harvesting e-addresses on the Internet.

- They find domain web sites through search engines and from direct links of a previous known site.
- They prospect common pages of a web site and rapidly stop when a "special" hyperlink appears.
- They gather e-addresses written in clear in HTML code. Some of them resolve ASCII protection and even basic javascript or flash code.
- They send e-addresses to common mailboxes directly to mail servers.
- They use e-address domains to find new sites.
- They trap reader with hyperlinks inside spams and with "confirmation of reading" requests to validate the e-address.
- They heavily prospect newsgroups to gather information in email headers.
- They use spyware solutions to spy the user and collect interesting information.
- They search the Whois database to find registrant's e-addresses. Robust protections seem to be sometimes overcome.
- They do not check the evolution of e-addresses. You should try to protect your e-address when it first reaches the Internet.

Internet user should adapt his attitude to avoid spams. A not so complex protection for e-addresses on his homepage suffices, at least for the moment. This protection must be in place at the first design of the page. Also, when user posts in newsgroups, his e-address must be carefully chosen, especialy the mail header ones. A good email client, working in offline mode all the time, with the "confirmation of reading" turned off, seems to be the best solution. And of course, he must not answer spam neither use spammers marketing.

System managers and web site designers should prevent the utilization of common identities like "root" or "info" and they may use a special e-address when buying a domain name for whois concern.

Further readings

[20] *Caspian Networks & CacheLogic. Capitalizing on the p2p opportunity. Available online at* `http://www.caspiannetworks.com/files/CaspianCacheLogic_P2P.pdf`*, October 2004.*

[68] *Brian McWilliams.* Spam Kings*. O'Reilly, 1st edition, 2004.*

Conclusions

CHAPTER VIII

Conclusions and Perspectives

In this dissertation, we tackled two different problems. In the large first part, we analyzed new side channels that exploit the memory architecture of current processors and proposed countermeasures. And in the second part, we studied the early stages of spam to propose effective countermeasures against the gathering of email addresses.

We now skim through our contributions in order to determine potential further works.

By nature, the cache mechanism presents different timings in the memory accesses: in the best case, the request data is hold by the cache and the access time is short (cache-hit), otherwise the data must be fetched from a higher memory level and the access time is longer (cache-miss). In cache-based attacks, a program with some secret data is executed and an attacker exploits the timing differences in order to discover this secret data.

In Chapter IV, we showed that the overall execution time of the program (with the secret data) is analyzed. We demonstrated the origin of the leakage in the case of AES and explained the link between the number of discovered key bits and the cache architecture. Moreover, we showed that this number is theoretically limited. However, this topic is far from being completely understood. Further research vectors can be distinguished:

- We considered random plaintexts. However, our preliminary analysis showed that chosen-plaintexts reveal patterns with more narrow sets of peaks, increasing the number of discovered bits. This is probably due to internal collisions. Defining critical plaintexts to encrypt can help lowering down the number of required measurements.
- One threat with time-driven attacks is its remote application (on real protocol). We showed that the problem is that the variation in the network transmission is too big. Therefore, one should focus on improving the signal to noise ratio of the encryption times. Following Bernstein's intuition, increasing the length of the plaintext should help.

- We also believe that high-class statistical methods can lower down the number of required measurements. A refined model of the system and the noise would be helpful.

In Chapter V, we focused on access-driven attacks. In this case, the attacker manages to execute another process during the encryption. This kind of attack then requires a synchronization with the crypto process. However, it is much more powerful than time-driven ones. We showed that the analysis of the last round, with respect to the ciphertext, outperforms other attacks since less than 15 measurements are needed to disclose a full 128-bit AES key, in noiseless conditions.

We proposed software mitigations in Chapter VI. They can be adapted to match the security needs and therefore limit the performance drawback to its limit. Our mitigations are the first concrete countermeasures to defend against AES cache-based attacks.

In those Chapters, we particularly focused our attention on AES, because it is the most commonly used one. However, many other ciphers exist and some of them might use the cache at one point or another. Software engineers should reconsider their security-related codes to make sure that no information leaks through the cache or that the signal to noise ratio is small enough. Software mitigations in the vein of ours should counter any cache-based vulnerabilities (with limited resolution). However, attackers with high resolutions could still be able to get some part of the secret. But so far, no high resolution attack has been presented.

This part takes its interests out of the possible applications in the framework of Trusted Computing and Isolation or Virtualization assumptions. Attacks as the ones we developed might jeopardize the trust chain, at the hardware level. The work must as well be put into the perspective of the level of sharing in the different multiple-core architectures. Also, although this thesis is focused on general-purpose microprocessors, a research fields is to apply cache-based techniques in other processors as for high-end smart cards or from other manufacturers.

On the other hand, the cache architecture is the first feature of the processor being attacked by side channels, with concrete results. As the processors are actually not designed from the point of view of side channels, other parts of the processor will be under attack. For that reason, a careful review of the processor's features must be done in short terms and countermeasures must be proposed reviewed and implemented at different levels. But, on a long term perspective, it is crucial to educate designers and developers to integrate security aspects at design time and even pre-design time.

Also, thorough works formalizing the power of potential attackers must be undertaken, in order to clearly determine the security threats and therefore adapt the countermeasures.

Wirth's law [15] states that *Software get slower faster than Hardware gets faster*. It motivates designers for software optimizations. However, one should not neglect the security vulnerabilities inserted by those optimizations. Increasing the performances is useless in a system where the such security considerations have been neglected.

The second part is composed of Chapter VII, which presents our study over the analysis of spam. The scope of our work was to quantify the quality of email address protection against the gathering tools of spammers. Moreover, we globally studied the behavior of the spammers concerning various specific points. After the three described phase in the Chapter, we extend our analysis to understand the principle that the spammers use to surf from one page to another and why they select an email address among many others. Moreover, our system keeps a record of the visit of the spammers and compute statistics on various properties. This new phase is currently running. More details will be given in a future publication.

Let us return to Alice, Bob and Eve. Thanks to the above Chapters IV and V, Eve got tools to infiltrate Alice and Bob's private conversation, through cache leakages. However, Bob received in Chapter VI fruitful means to counter again Eve's intrusions. So that is one battle won against side channels, but the war is certainly not over...

Bibliography

[1] Onur Acıiçmez, Werner Schindler, and Çetin K. Koç. Trace Driven Cache Attack on AES. e-print of the IACR, 2006. Available online at `http://eprint.iacr.org/2006/138.pdf`.

[2] AMD Corp. CPUID Specifications. Available online at `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25481.pdf`.

[3] AMD Corp. Software optimization guide for AMD Athlon^{TM}64 and AMD Opteron^{TM}processors, September 2003. Available online at `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF`.

[4] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289, New York, NY, USA, 2003. ACM Press.

[5] L. A. Belady. A study of replacement algorithms for a virtualstorage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. Available online at `http://cr.yp.to/papers.html\#cachetiming`.

[7] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES Power Attack Based on Induced Cache Miss and Countermeasure. In *ITCC (1)*, pages 586–591. IEEE Computer Society, 2005.

[8] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. *New trends in cryptographic systems: A Power Attack Methodology to AES Based on Induced Cache Misses: Procedure, Evaluation and Possible Countermeasures*, chapter 3, pages 37–52. Intelligent System Engineering. Hauppauge, NY : Nova Science Publishers, 2006.

[9] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[10] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, Baltimore, MD, August 2005.

[11] Eli Biham and Orr Dunkelman. Cryptanalysis of the a5/1 GSM stream cipher. In *INDOCRYPT*, pages 43–51, 2000.

[12] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. A new CRT-RSA algorithm secure against bellcore attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 311–320, New York, NY, USA, 2003. ACM Press.

[13] D. Boneh and D. Brumley. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[14] Christopher G. Bookholt. Address space layout permutation: Increasing resistance to memory corruption attacks. Master's thesis, North Carolina State University, 2005.

[15] László Böszörményi, Jürg Gutknecht, and Gustav Pomberger, editors. *The School of Niklaus Wirth, "The Art of Simplicity"*. dpunkt.verlag/Copublication with Morgan-Kaufmann, 2000.

[16] Phil Bradley. The spam experiment. `http://www.philb.com/spamex.htm`, October 2002.

[17] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Implementing crypto algorithms to defend against software side channels. Rump session of Crypto'05.

[18] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, 1996.

[19] Business Software Alliance. 1 in 5 british consumers buy software from spam. `http://www.bsa.org/uk/press/newsreleases/online-shopping-tips.cfm`, December 2004.

[20] Caspian Networks & CacheLogic. Capitalizing on the P2P Opportunity. `http://www.caspiannetworks.com/files/CaspianCacheLogic_P2P.pdf`, October 2004.

[21] Center for Democracy & Technology. Why am I getting all this spam? Unsolicited commercial e-mail research six month report. `http://www.cdt.org/speech/spam/030319spamreport.shtml`, March 2003.

[22] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Trans. Comput.*, 53(6):760–768, 2004.

[23] Pawel Chodowiec and Kris Gaj. Very Compact FPGA Implementation of the AES Algorithm. In Walter et al. [127], pages 319–333.

[24] Coalition Against Unsolicited Commercial Email. Why UCE is a threat to the viability of internet email and a danger to Internet commerce. `http://www.cauce.org/about/problem.shtml`.

[25] Joan Daemen and Vincent Rijmen. *The design of Rijndael, AES - The Advanced Encryption Standard*. Information Security and Cryptology. Springer, 2001.

[26] Theo de Raadt. Exploit Mitigation Techniques (in OpenBSD, of course). Available online at `http://www.openbsd.org/papers/ven05-deraadt/`.

[27] Franck Delattre. The AMD K8 Architecture, 2004. Available online at `http://www.cpuid.com/reviews/K8/index.php`.

[28] Peter J. Denning. The working set model for program behaviour. *Commun. ACM*, 11(5):323–333, 1968.

[29] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In Jean-Jacques Quisquater and Bruce Schneier, editors, *CARDIS*, volume 1820 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 1998.

[30] Network of Excellence in Cryptology ECRYPT. AES Lounge. Available online at `http://www.iaik.tugraz.at/research/krypto/AES/`.

[31] Federal Trade Commission. The CAN-SPAM Act: Requirements for commercial emailers. `http://www.ftc.gov/bcp/conline/pubs/buspubs/canspam.htm`, January 2004.

[32] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 67, Washington, DC, USA, 1997. IEEE Computer Society.

[33] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. pages 251–261, 2001.

[34] Gartner Inc. The spam within: Gartner says one-third of business e-mail is “occupational spam”. `http://www.gartner.com/5_about/press_room/pr20010419b.html`, April 2001.

[35] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[36] Gary Graunke, Jean-Pierre Seifert, and Ernie Brickell. Mitigating software side channels in AES and RSA software. DEV-203 of the RSA Conference 2006.

[37] Grant Gross. Internet service provider awarded $1 billion in spam damages. IDG News Service, `http://www.pcworld.com/news/article/0,aid,119011,00.asp`, December 2004.

[38] Jim Handy. *The cache memory book (2nd ed.): the authoritative reference on cache design*. Academic Press, Inc., Orlando, FL, USA, 1998.

[39] Alejandro Hevia and Marcos Kiwi. Strength of two data encryption standard implementations under timing attacks. volume 2(4), pages 416–437, New York, NY, USA, 1999. ACM Press.

[40] Mard D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, Computer Science Division (EECS), University of California, Berkeley, California, USA, 1987.

[41] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. (Q1):13, February 2001.

[42] Wei-Ming Hu. Lattice scheduling and covert channels. *Proceedings of the IEEE Symposium on Security and Privacy*, 25:52–61, 1992.

[43] IEEE Standards. IEEE 802.11i. Available online at `http://standards.ieee.org/getieee802/download/802.11i-2004.pdf`.

[44] Intel Corp. Using the RDTSC Instruction for Performance Monitoring. Available online at `http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM`.

[45] ITFacts. E-mail statistics. `http://www.itfacts.biz/index.php?id=P1475`, September 2004.

[46] Richi Jennings. The global economic impact of spam. Technical report, Ferris Research, `http://www.ferris.com/`, 2005.

[47] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2/3), 2000.

[48] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX(1):5–38, January 1883.

[49] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX(2):161–191, February 1883.

[50] Angelos D. Keromytis and Vassilis Prevelakis. A survey of randomization techniques against common mode attacks. Technical report, Department of Computer Science, Drexel University, 2005. Available online at `http://www.cs.drexel.edu/static/reports/DU-CS-05-04.pdf`.

[51] KillerSites.com. Anti-spam, protect your email address. `http://www.killersites.com/webDesignersHandbook/emailObscucator.htm`, 2005.

[52] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 3rd edition edition, 1997.

[53] 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.

[54] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[55] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[56] Cédric Lauradoux. Collision attacks on processors with cache and countermeasures. In *Christopher Wolf, Stefan Lucks, Po-Wah Yau (eds.) Proceedings of Western European Workshop on Research in Cryptplogy (WeWorc 2005). GI-Edition - Lecture Notes in Informatics (LNI), P-74, Bonner Köllen Verlag (2005)*, 2005.

[57] Robert Lemos. Crooks behind more net attacks. `http://news.com.com/Report+Crooks+behind+more+Net+attacks/2100-7349_3-5455225.html`, November 2004.

[58] Steven B. Lipner. A comment on the confinement problem. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 192–196, New York, NY, USA, 1975. ACM Press.

[59] Keith Loepere. Resolving covert channels within a B2 class secure system. *SIGOPS Oper. Syst. Rev.*, 19(3):9–28, 1985.

[60] Joe Loughry and David A. Umphress. Information leakage from optical emanations. *ACM Trans. Inf. Syst. Secur.*, 5(3):262–289, 2002.

[61] Joe Maller. Javascript: Protect your email address. `http://www.joemaller.com/js-mailer.shtml`, May 2001.

[62] Stefan Mangard, Manfred Aigner, and Sandra Dominikus. A Highly Regular and Scalable AES Hardware Architecture. *IEEE Trans. Computers*, 52(4):483–491, 2003.

[63] Mitsuru Matsui. How far can we go on the x64 processors? In *To appear in Proceedings of FSE 2006.*

[64] Mitsuru Matsui. New block encryption algorithm misty. In Eli Biham, editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 1997.

[65] Mitsuru Matsui and Sayaka Fukuda. How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[66] Máire McLoone and John V. McCanny. High Performance Single-Chip FPGA Rijndael Algorithm Implementations. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2001.

[67] Máire McLoone and John V. McCanny. Single-Chip FPGA Implementation of the Advanced Encryption Standard Algorithm. In Gordon J. Brebner and Roger Woods, editors, *FPL*, volume 2147 of *Lecture Notes in Computer Science*, pages 152–161. Springer, 2001.

[68] Brian McWilliams. *Spam Kings.* O'Reilly, 1st edition, 2004.

[69] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography.* CRC Press, Boca Raton, Florida, 1996. Available online at `http://cacr.math.uwaterloo.ca/hac`.

[70] MessageLabs. Email security intelligenge report. Whitepaper, `http://www.messagelabs.com/emailthreats/intelligence/whitepapers/pdf/sixreport.pdf`, June 2004.

[71] MessageLabs. Average global ratio of spam in email scanned by MessageLabs. `http://www.messagelabs.com/img/eng/emailthreats/spam_large.jpg`, 2005.

[72] MIPS technologies. MIPS32 4Kc processor core datasheet, November 2004. Available online at `http://www.mips.com/content/Products/Cores/32-BitCores/MIPS324KFamily/ProductCatalog/P_MIPS324KFamily/productBrief`.

[73] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Math. Computation*, 44:243–264, 1987.

[74] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture - Complexity and Correctness*. Springer-Verlag, 2000.

[75] Name Intelligence Inc. Free email protection script. `http://www.whois.sc/info/webmasters/email-protection.html`.

[76] National Institute of Standards and Technology. ANSI C Reference Code V2.0 (October 24, 2000).

[77] National Institute of Standards and Technology (NIST). FIPS PUB 46, The Data Encryption Standard. Federal Information Processing Standard, NIST, U.S. Dept. of Commerce., 1977.

[78] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard. Federal Information Processing Standard, NIST, U.S. Dept. of Commerce, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.

[79] OpenSSL. OpenSSL: the Open-source toolkit for SSL / TLS. Available online at `http://www.openssl.org/`.

[80] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache atacks and countermeasures: the case of AES (extended version), 2005. Available online at `http://www.wisdom.weizmann.ac.il/~tromer/`.

[81] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and Countermeasures: the Case of AES. Cryptology ePrint Archive, Report 2005/271, 2005. Available online at `http://eprint.iacr.org/2005/271.pdf`.

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

[83] Out-Law. The spammers are watching you. `http://www.out-law.com/php/page.php?page_id=pressrele3360`, February 2003.

[84] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.

[85] Dan Page. Defending against cache based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, April 2003.

[86] Dan Page. Partitioned cache architecture as a side-channel defense mechanism. Cryptology ePrint Archive, Report 2005/280, 2005. Available online at `http://eprint.iacr.org/2005/280.pdf`.

[87] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition edition, 2003.

[88] Colin Percival. Cache missing for fun and profit, 2005. Available online at `http://www.daemonology.net/hyperthreading-considered-harmful/`.

[89] Norbert Pramstaller, Stefan Mangard, Sandra Dominikus, and Johannes Wolkerstorfer. Efficient AES Implementations on ASICs and FPGAs. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *AES Conference*, volume 3373 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2004.

[90] Norbert Pramstaller and Johannes Wolkerstorfer. A Universal and Efficient AES Co-processor for Field Programmable Logic Arrays. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *FPL*, volume 3203 of *Lecture Notes in Computer Science*, pages 565–574. Springer, 2004.

[91] B. Preneel, A. Biryukov, C. De Cannière, S. B. ¨Ors, E. Oswald, B. Van Rompay, L. Granboulan, E. Dottax, G. Martinet, S. Murphy, A. Dent, R. Shipsey, C. Swart, J. White, M. Dichtl, S. Pyka, M. Schafheutle, P. Serf, E. Biham, E. Barkan, Y. Braziler, O. Dunkelman, V. Furman, D. Kenigsberg, J. Stolin, J-J. Quisquater, M. Ciet, F. Sica, H. Raddum, L. Knudsen, and

M. Parker. Final report of European project number IST-1999-12324, named New European Schemes for Signatures, Integrity, and Encryption, April 2004.

[92] Matthew Prince. Project honey pot, `http://www.projecthoneypot.org/`. In Massachusetts Institute of Technology, editor, *Spam Conference*, January 2005. `http://www.spamconference.org/`.

[93] Jean-Jacques Quisquater and Chantal Couvreur. Fast Decipherment Algorithm for RSA Public-Key Cryptosystem. *Electronics Letters*, 21(18):905–907, 1982.

[94] Jan M. Rabaey. *Digital integrated circuits: a design perspective.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[95] RFC - Network Working Group. RFC 2246:The TLS Protocol (Version 1.0). Available online at `http://www.ietf.org/rfc/rfc2246.txt`.

[96] RFC - Network Working Group. RFC 2401: Security Architecture for the Internet Protocol; RFC 2402: Authentication Header; RFC 2403: The Use of HMAC-MD5-96 within ESP and AH; RFC 2404: The Use of HMAC-SHA-1-96 within ESP and AH; RFC 2405: The ESP DES-CBC Cipher Algorithm With Explicit IV; RFC 2406: Encapsulating Security Payload; RFC 2407: IPsec Domain of Interpretation for ISAKMP (IPsec DoI); RFC 2408: Internet Security Association and Key Management Protocol (ISAKMP); RFC 2409: Internet Key Exchange (IKE); RFC 2410: The NULL Encryption Algorithm and Its Use With IPsec; RFC 2411: IP Security Document Roadmap; RFC 2412: The OAKLEY Key Determination Protocol. Available online at `http://www.ietf.org/rfc/rfc24XX.txt`, where `XX` are the final digits of the RFCs.

[97] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. United States Patent 4405829: Cryptographic communications system and method. Available online at `http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=4,405,829`.

[98] 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.

[99] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *International Symposium on Microarchitecture*, pages 24–35, 1996.

[100] Sandpile. AA-64 implementation AMD K8. Available online at `http://www.sandpile.org/impl/k8.htm`.

[101] Sandpile. IA-32 implementation Intel P4 (incl. Celeron and Xeon). Available online at `http://www.sandpile.org/impl/P4.htm`.

[102] Patrick Schaumont, Henry Kuo, and Ingrid Verbauwhede. Unlocking the design secrets of a 2.29 Gb/s Rijndael processor. In *DAC*, pages 634–639. ACM, 2002.

[103] Werner Schindler, François Koeune, and Jean-Jacques Quisquater. Improving divide and conquer attacks against cryptosystems by better error detection / correction strategies. In Bahram Honary, editor, *IMA Int. Conf.*, volume 2260 of *Lecture Notes in Computer Science*, pages 245–267. Springer, 2001.

[104] Kai Schramm, Thomas J. Wollinger, and Christof Paar. A New Class of Collision Attacks and Its Application to DES. In Thomas Johansson, editor, *FSE*, volume 2887 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 2003.

[105] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

[106] Tom Shanley. *The Unabridged Pentium 4 : IA32 Processor Genealogy.* Addison-Wesley Professional, 2004.

[107] John Shen and Mikko Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.

[108] Olin Sibert, Phillip A. Porras, and Robert Lindell. The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems. In *SP '95: Proceedings of the 1995 IEEE Symposium on Security and Privacy*, page 211, Washington, DC, USA, 1995. IEEE Computer Society.

[109] Abraham Silberschatz, Greg Gagne, and Peter B. Galvin. *Operating system concepts (7th ed.)*. John Wiley and Sons, Inc., ?, ?, USA, 2005.

[110] Sophos Inc. Controversial Lycos europe anti-spam screensaver put on hold. `http://www.sophos.com/spaminfo/articles/lycos.html`, December 2004.

[111] Sophos Inc. Judge rejects guilty plea of AOL spam case suspect said to have stolen 92 million email addresses. `http://www.sophos.com/spaminfo/articles/aolreject.html`, December 2004.

[112] Sophos Inc. The spam economy: the convergent spam and virus threats. `http://www.sophos.com/sophos/docs/eng/papers/Sophos_spam-economy_wpus.pdf`, August 2004.

[113] Spamhaus. The register of known spam operations. `http://www.spamhaus.org/rokso/index.lasso`.

[114] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs. In Walter et al. [127], pages 334–350.

[115] Syronex. Safemailto: A strong anti-spam email address encoder. `http://w2.syronex.com/jmr/safemailto/`, April 2005.

[116] GCC team. GCC online documentation. Available online at `http://gcc.gnu.org/onlinedocs/`.

[117] PaX Team. address space layout randomization, March 2003. Available online at `http://pax.grsecurity.net/docs/aslr.txt`.

[118] Brad Templeton. Origin of the term "spam" to mean net abuse. `http://www.templetons.com/brad/spamterm.html`.

[119] Jonathan Trostle. Timing attacks against trusted path. In *In IEEE Symposium on Security and Privacy*, 1998.

[120] Trusted Computing Group. Trusted Computing Group. Available online ar `https://www.trustedcomputinggroup.org/home`.

[121] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. In Walter et al. [127], pages 62–76.

[122] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Miyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *Proceedings of International Symposium on Information Theory and Its Applications*, pages 803–806, 2002.

[123] Yukiyasu Tsunoo, Etsuko Tsujihara, Maki Shigeri, Hiroyasu Kubo, and Kazuhiko Minematsu. Improving cache attacks by considering cipher structure. *International Journal ofInformation Security*, 2005.

[124] United States Patent 4195343. Round robin replacement for a cache store.

[125] United States Patent 6138225. Address translation system having first and second translation look aside buffers.

[126] Arjan van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.3, update 3, August 2004. Available online at `http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf`.

[127] Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop,*

Cologne, Germany, September 8-10, 2003, Proceedings, volume 2779 of *Lecture Notes in Computer Science*. Springer, 2003.

[128] Washington State University. E-mail harvest protection. `http://www.wsu.edu/identity/web/resources/email-harvest.html`.

[129] Christopher Westley. The economics of spam. `http://www.fee.org/vnews.php?nid=5662`, November 2003.

[130] Wikipedia, the free encyclopedia. CPU cache. Available online at `http://en.wikipedia.org/wiki/CPU_cache`.

[131] Paul Wood. Save yourself from eternal spamnation. MessageLabs Whitepaper, `http://www.messagelabs.com/emailthreats/intelligence/whitepapers/pdf/SpamWhitePaper-US.pdf`, April 2004.

[132] John C. Wray. An analysis of covert timing channels. *Journal of Computer Security*, 1(3-4):219–232, 1992.

[133] Robert Hobbes Zakon. Hobbes' internet timeline. IETF RFC2235, `http://www.zakon.org/robert/internet/timeline/`, January 2005.

APPENDIX A

Publication list

A.1. Publications

(1) Lilian Bohy, Michael Neve, David Samyde and Jean-Jacques Quisquater, *Principal and Independent Component Analysis for Cryptographic systems with Hardware Unmasked Units*, in proceedings of e-Smart 2003, Sophia-Antipolis, France, September 2003.

(2) Michael Neve, Eric Peeters, David Samyde and Jean-Jacques Quisquater, *Memories: a Survey of their Secure uses in Smart Cards*, in proceedings of 2nd International IEEE Security In Storage Workshop (IEEE SISW 2003), Washington DC, USA, October 2003.

(3) Mathieu Ciet, Michael Neve, Eric Peeters and Jean-Jacques Quisquater, *Parallel FPGA Implementation of RSA with Residue Number Systems - Can side channel threats be avoided?*, in proceedings of IEEE Midwest 2003, Cairo, Egypt, December 2003.

(4) Jean-Jacques Quisquater, Michael Neve, Eric Peeters and François-Xavier Standaert, *L'émission rayonnée des cartes à puce : une vue d'ensemble*, Revue de l'Electricité et de l'Electronique, N° 6/7, Juin-Juillet 2004, pp. 78-81, Ed. Société de l'Electricité, de l'Electronique et des Technologies de l'Information et de la Communication, Paris, France.

(5) Eric Peeters, Michael Neve and Mathieu Ciet, *XTR Implementation on Reconfigurable Hardware*, in proceedings of Cryptographic Hardware and Embedded Systems (CHES 2004), Lecture Notes in Computer Science 3756, pages 386-399, Springer-Verlag, IACR, 2004.

(6) Mathieu Ciet, Michael Neve, Eric Peeters and Jean-Jacques Quisquater, *Parallel FPGA Implementation of RSA with Residue Number Systems - Can side channel threats be avoided? - Extended version*, IACR e-print, 2004.

(7) Damien Giry, Michael Neve and Jean-Jacques Quisquater, *e-Mail Address Protection Study*, in Short Paper Proceedings of

Information Security and Cryptology (CISC 2005), Higher Education Press, pp. 247-256, Beijing, China 2005.

(8) Michael Neve, Jean-Pierre Seifert and Zhenghong Wang, *A refined look at Bernstein's AES side channel analysis*, Fast Abstract Proceedings of AsiaCCS 2006, Taipei, Taiwan.

(9) Ernie Brickell, Gary Graunke, Michael Neve and Jean-Pierre Seifert, *Software mitigations to hedge AES against cache-based software side channel vulnerabilities*, IACR e-print, 2006.

(10) Michael Neve, Eric Peeters, Guerric Meurice de Dormale and Jean-Jacques Quisquater, *Faster and smaller hardware implementation of XTR*, Invited talk in Advanced Signal Processing Algorithms, Architectures, and Implementations XVI, in Proceedings of SPIE Symposium on Optics & Photonics, San Diego, California, USA, August 2006.

(11) Michael Neve and Jean-Pierre Seifert, *Advances on Access-driven Cache Attacks on AES*, Accepted to Selected Areas of Cryptography (SAC 2006), Montreal, Canada, August 2006.

A.2. Under submission

(1) Early Stages of SPAM: First Quantitative Analysis

(2) XTR: State of the Art in software and hardware implementations, side-channel and fault analysis.