POLITECNICO
MILANO 1863

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in Computer Science and Engineering

---

Information-Leakage Analysis based on Hardware Performance Counters

Master Thesis of:
Matteo Maria Fusi
Matr. 849803

Advisor:
Prof. Cristina Silvano
Co-Advisor:
Prof. Alessandro Barenghi

Academic Year 2016–2017

Ringraziamenti

Con questo lavoro si conclude un capitolo della mia vita e se ne aprirà uno nuovo. Mi sembra quindi giusto spendere qualche ringraziamento per chi mi ha permesso di concludere il mio percorso di studi.

Il primo e più speciale di tutti va a Carmen che in questi dieci lunghi mesi di lavoro di tesi ha sempre creduto in me anche quando io stesso non ero sicuro dei miei mezzi.

Grazie anche alla mia famiglia che mi ha sempre insegnato che l'impegno ed i sacrifici vengono ripagati; perciò grazie Mamma, Papà ed Emanuele.

L'ultimo ringraziamento va a tutte le persone che ho incontrato nell'ambiente scolastico: dai bambini piagnucolosi delle scuole elementari agli studenti secchioni dell'università, dai docenti buoni a quelli cattivi, dagli amici nuovi a quelli vecchi, da chi mi ha aiutato a chi mi ha messo in difficoltà. Vi ringrazio perchè ognuno di voi, poco o tanto, mi ha reso chi sono ora e mi ha perciò permesso di raggiungere la fine di questo faticoso percorso.

Grazie di cuore,

Matteo

Abstract

Hardware performance monitoring allows to easily control hardware events at process or system level. This system makes possible to track a great variety of events or measures related to the system: from the number of executed instructions to the energy consumption of the system. If the access to such a system is not properly controlled, it may reveal information about the execution of sensitive processes such as cryptographic routines. This document proposes a methodology to verify if a target hardware performance counter leaks information about the instructions executed by the processor. This analysis is executed building a set of simples test-cases that can influence such counters and performing simple statistical analysis. Moreover, this document implements this methodology on an Intel processor and it verifies if the Intel RAPL interface leaks information. Intel RAPL reports the energy consumption of the package, the cores and DRAM at system level. On Linux systems, these counters can be potential side-channel and this document verifies this fact by performing a case study on RSA-16384. The result is that the low resolution of these counters limits the effects of a potential side-channel attack.

Estratto

Il monitoring delle prestazioni attraverso i contatori hardware permette di monitorare in modo semplice eventi hardware sia a livello di processo che di sistema. Questo sistema rende possibile il controllo di una grande varietà di eventi o misure del sistema: dal numero di istruzioni eseguite al consumo energetico. Se l'accesso a questa infrastruttura non è controllato in maniera appropriata, esso potrebbe rivelare informazioni riguardante l'esecuzione di processi come le implementazioni software di cifrari. Questo documento propone una metodologia per verificare se un contatore hardware delle prestazioni soffre di perdita di informazione riguardante le instruzioni eseguite dal processore construendo un semplice insieme di casi test che possono influenzare tale contatore e utilizzando l'analisi statistica. Inoltre, questo documento implementa la metodologia citata su un processore Intel e verifica se l'interfaccia Intel RAPL soffre di perdita di informazioni. L'interfaccia Intel RAPL riporta i consumi energetici del package, dei cores e della DRAM a livello di sistema. Su sistemi Linux questi contatori possono essere un potenziale side-channel e questo documento verifica questo fatto eseguendo un caso di studio su RSA-16384. Il risultato è che la bassa risoluzione di questi contatori limita l'effetto di un potenziale attacco side-channel.

Contents

List of Figures

List of Tables

List of Algorithms and Listings

List of Acronyms

ASLR Address Space Layout Randomization

B Bytes

clk Clock cycles

CRT Chinese Remainder Theorem

FIVR Fully-Integrated Voltage Regulator

FR Flush+Reload

LSB Least Significant Bit

MPI Multiple Precision Integer

MSR Model Specific Register

MSB Most Significant Bit

OS Operating System

PC Program Counter

RAPL Running Average Power Limit

ROB ReOrder Buffer

SCA Side Channel Attack

Chapter 1

Introduction

This chapter offers to the reader an introduction to the problem and the goals that this document wants to solve. Also the structure of this work is present. Section 1.1 introduces the reader to the possible information-leak problems that hardware performance counters may introduce in modern processors and Section 1.2 displays the organization of this work.

1.1 Problem and Motivations

Hardware performance monitoring allows to easily monitor hardware events at process or system level system by simply configuring a set of specific registers. It is possible to monitor a great variety of events or statistics: from the number of executed instructions to the energy consumption of the system. If hardware performance monitoring is not limited, it may reveal information about the execution of sensitive processes such as cryptographic routines.

1.1.1 Objectives

This study wants to analyze the possibility of an information-leakage through hardware performance counters. The information-leakage can be verified by building a test framework based on a simple test-case methodology. Moreover, this document wants to discover if it is possible to use the found infor-

mation from the previously mentioned methodology to verify if it is possible to leak information by a cryptographic application. In a more structured manner, this document has the following goals:

1. Propose a methodology that verifies if a specific hardware counter leaks information about a set of specified operations.

2. Propose an implementation of the methodology focusing on the analysis of Intel RAPL counters on an Intel Broadwell-U processor.

3. Use the information provided by the methodology to conduct a case study on the cited architecture.

The RAPL interface is composed by a set of hardware counters that Intel offers to monitor energy consumption. Point 1 is present in Chapter 4, while the remaining points are described in Chapter 6. The case study is focused on the analysis of a sign routine of RSA-16384 implemented with *libgcrypt* 1.7.6 and it wants to perform a Side-Channel analysis which recalls both from Flush+Reload and Simple Power Analysis.

1.2 Thesis Structure

Chapter 2 introduces the reader to the context of the research with a review of the target architecture (Intel *x86*) and Side-Channel attacks. Chapter 3 illustrates the current state of the art of cache Side-Channel attacks focusing on the Flush+Reload technique and the last studies related to Intel RAPL counters. Chapter 4 displays the proposed methodology to verify if a target hardware counter leaks information about the execution of specific instructions. Chapter 5 describes the tools used, how they're implemented and the problems related to the measurements in the target system. Chapter 6 shows the experimental results of the methodology implemented with the tools previously described on an Intel Broadwell-U processor and it proposes a case-study that wants to use the results obtained from the proposed methodology to extrapolate information from an application. Chapter 7 quickly reviews the proposed work and expresses final conclusions.

Chapter 2

Background

In this chapter, the reader will be introduced to the background of this document. The most relevant details of the target Intel *x86* architecture, the page sharing optimization implemented in modern OSs (2.2) and a quick introduction to SCAs (2.3) are displayed. Section 2.1 describes the *x86* architecture, and the related subsections shows more in detail the memory system (2.1.1), hardware performance counters (2.1.2) and the Intel RAPL interface (2.1.3). The most important concept of this chapter is to understand how pages and caches are shared in modern systems, since is the requirement to understand how a Flush+Reload attack works.

2.1 Target Architecture: Intel *x86*

The target architecture is Intel *x86*. It is not feasible to describe the entire architecture in one thesis, so only the relevant aspects are described. The main goal of the processor is to perform operations on memory by running programs. A memory is a physical device capable of storing information. Memory is addressed, which means that it is possible to map its content uniquely using a number called address. The memory used by the processor resides in a DRAM component, which is off-package. The package is interconnected to the main memory with a bus system. The processor can execute a set of architecure-dependent operations called instructions. The

set of the available instructions in a specific architecture is called Instruction Set Architecture (ISA).

The processor offers one or more cores to the programs. The cores are, in fact, responsible of effectively executing instructions issued by the programs. A complex implementation of the cores allows to executes more than one instruction at the time out-of-order and it applies the result in-order by using a ReOrder Buffer (ROB). It is possible to serialize the execution with some specific instructions [28]. The execution of the operation in the cores relies on the fact that it is possible to store and change copies of data present in main memory in registers. Registers are very-small, but high-performant memory embedded in the cores.

The set of available instructions also depends on the mode the architecture is running. The modes are the following ones [28]:

Protected mode This mode is the native state of the processor. Among the capabilities of protected mode is the ability to directly execute "real-address mode" 8086 software in a protected, multi-tasking environment.

Real-address mode This mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.

System management mode (SMM) This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security.

The focus of this document is on the protected mode, which is the state associated to any normal running program. The protected mode implements a system of rings that limits the execution of the instruction to counter malicious behaviours. The rings are the following ones:

0 Kernel Operations that can alter the hardware configuration such as page handling, or MSR operations.

1,2 Device Driver Not used.

3 User Perform non-privileged operations.

The kernel ring is the one that has access to the most-privileged instructions. User ring cannot use the set of instructions associated to lower levels, but there are mechanism to run specific routines with higher-privileges. An example of this case is the *SYSCALL* instruction [28]. Rings 1 and 2 are not used and the code runs in kernel mode or in user mode. Main memory is managed directly by the processor in kernel mode and its atomic unit is called page.

At hardware level it is possible to see a process as all the pages related to it and the current state of the execution. Pages contains both instructions related to a program and memory regions used to store results of the execution. The cited state is stored and updated in a special register called Program Counter (PC). The PC contains the address to a portion of memory that contains the instructions that will be fetched from the main memory. Two processes may share a single core if the Hyper-threading technology is enabled.

2.1.1 Memory System

Since the cores operate at a speed greater than the one associated to memory, a system of small but high speed memories called caches were introduced between the main memory and the cores. This system contains copies of data that resides in the main memory. Such a system allows to save fetching time from main memory in case a specific information is still saved in the cache system. The cache system is organized in a hierarchy: caches near the core are numbered with low numbers (high levels) w.r.t caches near the main memory (lower levels). See Figure 2.1.1 for an example. By convention, the cache that communicates directly with the core is the cache at Level 1. By descending toward the main memory, the number associated to the cache level increases. High levels of caches (i.e. near to the core) has lower dimension and higher speed w.r.t caches at low levels. Depending on the

level, the cache may be shared among cores or they can belong exclusively to one core. Cache line is the atomic unit of caches and it is dimensioned big enough to increase spatial locality. A cache line holds one aligned, power-of-two-sized block of adjacent bytes loaded from memory. If any byte in the line needs to be replaced, the entire line is substituted by a new one. Since the dimension of a cache is much lower than main memory, cache lines that originate from different regions of main memory are mapped on a single cache line. The dimension of the cache line may change among the levels of cache.

A cache works in the following way: when the core requests a specific data, it fetches the cache line associated from a lower level of cache (or directly from the memory in case it is the last level) and it stores it. The lines resides in the inquiring cache until it is replaced by another cache line which is addressed in the same way of the previous one. A cache line may also cleared by invalidation (also called eviction). If the core requests an information contained in the cache system, then a cache hit takes place. If the information is not present, then a cache miss takes place. In case of a cache miss, the operation that fetch the information and put it in the cache system is called cache refill [23]. A smart hardware may anticipate the fetch of a cache line with a prefetching system.

The policy that decides how lines are placed in cache is a key point in designing performant caches. The main used policies are the following ones [23]:

N-Way Set Associative An information is mapped onto a set in which is placed. A set corresponds to a group of N lines in the caches.

Direct-Mapped It is the case of N-way set associativity with one line per set.

Fully Associative The cache contains just one set. This mean that a cache line can be placed in every region of the cache.

The lines are addressed within a set by using a *tag,* which is simply an address derived by the original one.

In case of a cache hierarchy with more than one level of cache it is also important to define the policy of inclusion. If the block contained in a higher level of cache are surely contained in the lower levels, then the lower level of cache is inclusive of the higher level. If the lower level cache contains blocks that are not present in the higher level cache, then the lower level cache is said to be exclusive of the higher level cache [44]. If a line must be updated (due to modifications of the core execution) there are two possible way to update the lower levels of cache [23]:

Write-Back Update the lower levels of cache when the cache lines must be replaced or evicted.

Write-Through Update lower levels of cache immediatly after the cache lines is modified.

Modern Intel architectures split the L1 cache in Instruction and Data cache. Obviously, Instruction cache contains only cache lines associated to instructions, while the Data cache contains everything else. This feature allows to increase the instruction fetch bandwidth, so increasing the number of instructions executed per second. Before Nehalem, Intel processors had only two levels of cache, but the release of this architecture an additional third level (L3) was introduced. The last level of cache is usually shared between cores, while the others are related to a single core (but they may be shared between processes that runs on the same core) [15]. Intel uses any sort of cache-prefetching to speed up the computation. Instructions that will be executed after the one pointed by the PC may be prefetched directly by the hardware [28].

The address space that a process uses is virtual. Process memory is an abstraction used to simplify programming. The problem is that the hardware uses physical addresses, so there's a hardware component responsible of mapping virtual and physical addresses, it is called Memory Management Unit (MMU). The mechanism implemented by the MMU is called address translation and it is implemented in hardware and it is totally transparent for the process[28]. The MMU mechanism is put between the L1 and L2: L1 caches uses virtual addressing, while lower levels use physical ones.

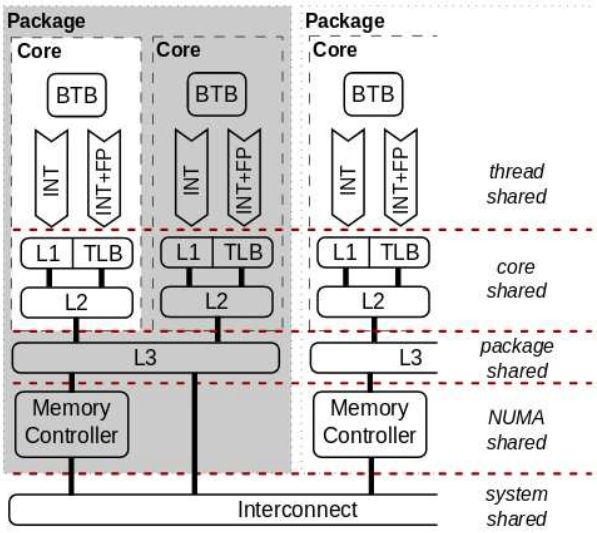

Figure 2.1.1: Memory organization of an Intel Core from Nehalem up to now. Note that different cores share the main memory and the L3 cache.

In some cases, programmers can partially manipulate cache system to increase performance of processes: an example is the *CLFLUSH* instruction. This instruction forces the eviction of the cache line from all the levels of the cache hierarchy of the cache lines associated to the virtual address passed as argument. The usage of this instruction is intented to free space in cache easily from process-side, so it is possible to use this instruction in user mode without any restriction [28].

2.1.2 MSRs and Hardware Performance Counters

The complexity of the architectures greatly increased over the years. This fact encouraged processors' producers to introduce a set of specific register to control and monitor the system. These counters are called Machine Specific Registers (MSRs) and they're present in every modern processor. Intel introduced the feature of MSRs with Pentium processors in the '90s [28]. A MSR which has a certain function may have different addresses in different architectures, so handling MSR is highly architecture-dependant and requires a detailed knowledge of the specific architecture and the related documen-

tation. In Intel processors, reading and writing MSRs must be performed using respectively *RDMSR* and *WRMSR* instructions in kernel mode.

A particular class of MSRs is the one that implements hardware performance monitoring. The registers associated to this class are called Performance Monitor Counters (PMCs). This system allows to monitor a defined set of micro-architectural details such as cycles, cache hit or misses, μops executed. The advantage of PMCs is that it is not necessary to build complex software instrumentation systems for performace monitoring and values can be collected in almost zero-time w.r.t software solutions. Every monitorable event has a specific code (that can varies between different architectures) that must be used to program specific MSRs of the class *IA32_PERFEVTSELx,* where x is a number that indicates the programmed PMC. The number of PMCs changes among different architectures.

2.1.3 The Intel RAPL Interface

Since also energy consumption became important to design software, in 2011 Intel introduced the Running Average Power Limit (RAPL) interface in SandyBridge. RAPL is composed by a set of MSRs that monitors and regulates energy consumptions of the system. The RAPL system allows monitoring and managing of the following zones:

Package Control and monitor the whole package.

DRAM Control and monitor the power consumption of the memory. This power zone was available only for cores designed for server applications, but from Haswell it was made available also for "everyday-use" processors [14].

Cores Control the power consumption of the cores.

Off-core Control the power consumption of an off-core element. The monitored component is architecture dependant.

The counters are updated with a period of $976\mu s$ and they have a sensitivity that depends on the architecture and on the power zone. To know

the sensitivity the MSR *MSR_RAPL_POWER_UNIT* must be controlled [26, 14, 28]. Before Haswell, RAPL measurements were produced with a modeling approach trained during the boot phase of the system, but now they are produced by directly probing the die [20] thanks to the probes associated to the Fully-Integrated Voltage Regulator (FIVR) system; but it is not clear how really the RAPL counters are produced in all the Intel architecture because of a lack of details in Intel Documentation [14]. RAPL counters are not officially considered part of the Intel Performance Monitoring unit because they're mainly related to energy capping, but they, anyway, offer a hardware value of energy consumption.

2.2 Operating System and Page Sharing

The Operating System (OS) is the software responsible of interfacing programs with the hardware. While normal programs use user code (Ring 3), the OS performs most of its operations in kernel mode (Ring 0). The OS is also responsible of mantainig the environment of every process safe to counter malicious behaviour of users of the system. The most simple measure is isolation. This principle should guarantee that a process cannot modify the behaviour of another one.

The OS has control over pages and it performs optimizations for the sake of improving the system performance. One of them is called page sharing and it is a method to save memory: if the OS finds that two processes can use the same page, then it just use one page instead of two identical copies. The shared page is obviously secured against the alteration: it is set read only and/or copy-on-write is allowed. Shared libraries are, in fact, based on the principle of page sharing: when the OSs loads two programs that uses the same shared library, it maps the same page. This features is present in OSs from more than 30 years [17, 39]. Combining the information of this section with Section 2.1 it is possible to observe the interaction between software and hardware: the OS maps virtual memory of the processes to pages. A part of the data contained in pages is replicated in the cache system to speed up memory access of the CPU. Figure 2.3.3 is a representation of the described

interaction. The green page is shared between two processes running on different cores. They share the cache line in L3, but they have their private copies in L1 and L2.

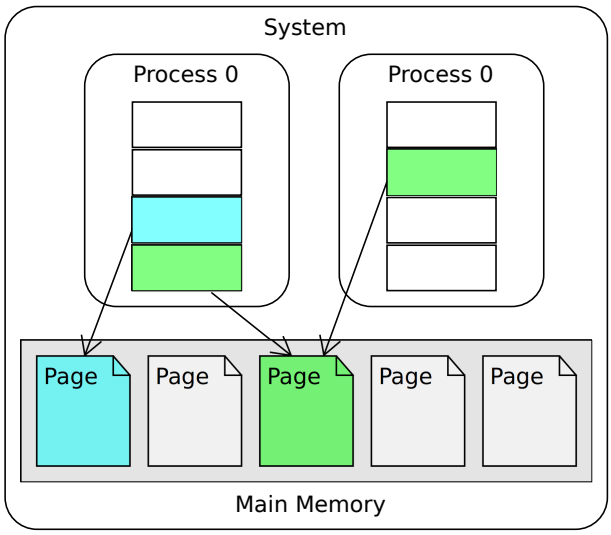

Figure 2.2.1: Two processes can share a page: this is an optimization performed by the OS to save memory.

2.3 Side Channel Attacks

Side Channel Attack (SCA) is a tecnique which extracts sensitive information by observing side channels. Considering a perfect cryptographic device, if you give it a clear text and a key it returns instantly the cipher text.

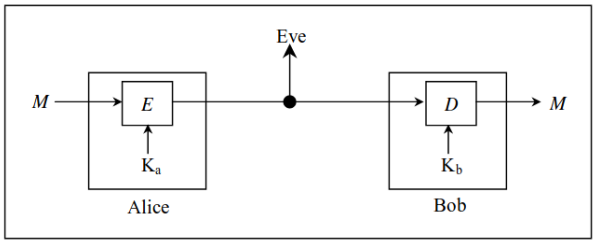

Figure 2.3.1: An ideal exchange of messages with perfect devices between Alice and Bob. Eve cannot know what the devices of Alice and Bob are doing. Source: [51].

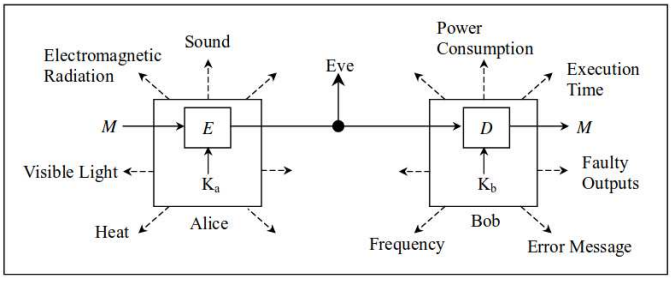

Figure 2.3.2: Classic cryptographic message exchange with real devices. Eve could observe side-channels, which are indicated by dashed lines, to know what the devices are doing. Source: [51].

In this (ideal) world side-channels does not exist, but in the real world systems are not perfect, so they consume power, emit electromagnetic-waves and require time to produce the output and compute intermediate results during computation. If one of these "real world" effects can be observed, then it is possible to observe a side-channel. Side-channels can be usefully exploited by an attacker if there's a sort of correlation between the output of the side-channel w.r.t what a device is doing. If this is the case, then it is possible to affirm that the channel *leaks* information. From the point of the attacker, SCA requires to know very well the structure of the monitored device, so SCAs targets specific hardware and/or software implementations.

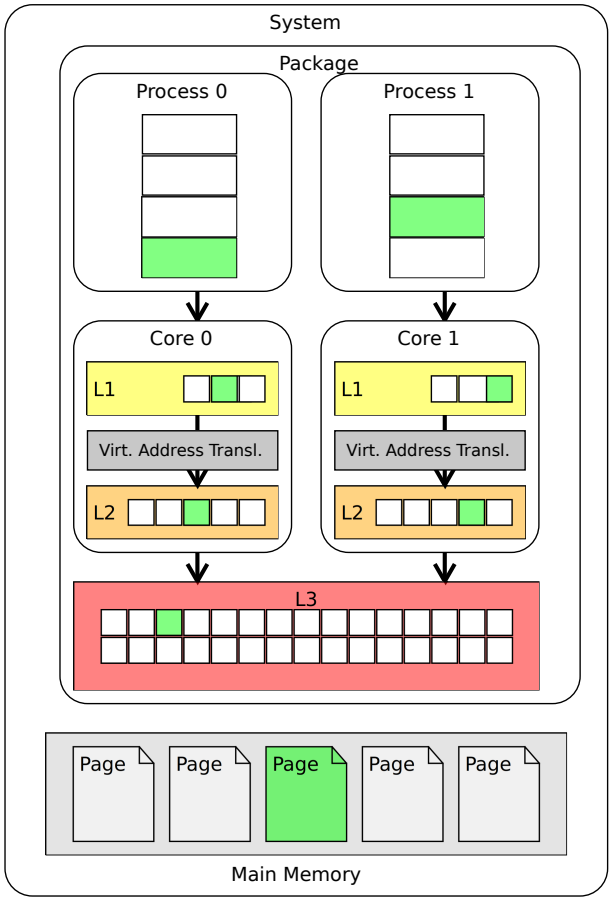

Figure 2.3.3: Virtual memory is mapped into pages replicating a part of data in the cache system.

Chapter 3

State of the Art

This Chapter describes the recent studies on cache SCAs, especially focusing of Flush+Reload attacks. Also studies on RAPL counters are enunciated. Section 3.1 offers a quick overview of Simple Power Analysis and the most common cache SCAs. Section 3.2 is a description of how it is composed a Flush+Reload attack and all the recent works about this technique are collected. Section 3.3 displays the recent studies about the Intel RAPL interface.

3.1 Side Channel Attacks

Between all the SCAs the most simple one is Simple Power Analysis (SPA), which was also one of the first side-channel attacks ideated by Kocher in 1999 alongside Differential Power Analysis (DPA) [32]. SPA and DPA were the first SCAs that had practical impact. SPA uses the trace of the consumed power of the target device as side-channel. This attack does not require a great number of traces and it is suitable in case the attacked algorithm leaks data-dependant information on the side-channel. An example power trace suitable for power analysis in shown in Figure 3.1.1. The proposed figure is a power trace of an encription performed with RSA using a binary modular exponentiation. As will be described in the case-study section (6.3), the encryption of RSA is key-dependant and it allows to easily observe 0 and 1

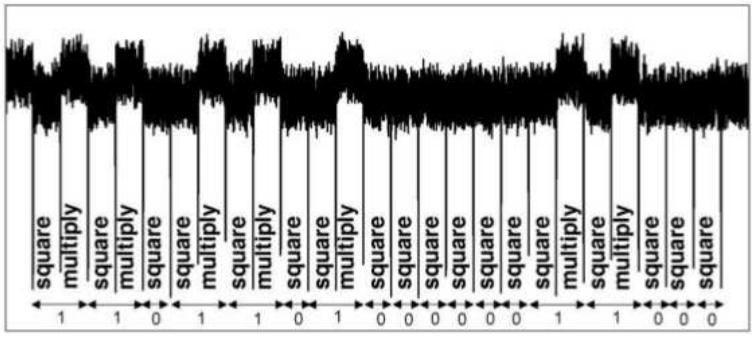

Figure 3.1.1: A power trace where an RSA encryption is running using simple binary modular exponentiation [41].

bits (like in this case) if the implementation is not carefully designed in a secure way. In this case it is simple to apply SPA knowing how much time is required to perform a modular multiplication and knowing that a muliply consumes more than a square.

Cache attacks are an important class of SCAs. The origins of these attacks comes from work of Page [42] based on Kelsey et al. [31]. The idea is to use caches as side channel by observing the access pattern of a victim process.

3.1.1 Cache Side Channel Attacks

As stated in Section 3.1, Page put the foundations of cache SCA with [42]. Tsunoo et al. produced a practical implementation in [45]. All these attacks have in common the fact that they all exploit cache-sharing and the possibility to control it in someway at user level. This attacks are considered of low/medium [1] severity because they're considered hard to implement, but it was demonstrated several times that it is possible to build both cross-core or cross-VM attacks ([50][49] [25]), and these attacks are valid also on ARM architectures [35]. These are the main cache channel attacks at the time of writing this document:

Evict+Time The attacker causes the victim process to run (this ensures that memory blocks are loaded in cache) , then it evicts target lines and causes a new execution of the target process. A variation in execution

time may leak information about the target process. This attack was ideated by Osvik et al. [40] while Bernstein elaborated a similar attack [9].

Prime+Probe This attacks refills target cache sets (which are also used by the target process) with its own lines and waits. Once the victim has executed, the attacker tries to access the lines loaded previuosly. If the lines of the malicious process were evicted, then it means that the victim process has performed a particular (monitored) operation [40][36].

Flush+Reload It is the inverse of Prime+Probe, but it is possible to target a specific cache line insead of an entire set. Section 3.2 gives a more focused overview on Flush+Reload state-of-the-Art attacks.

Flush+Flush It is an attack similar to Flush+Reload and it wants to be stealthier than its brother and it has an higher frequency, but it is more noisy. The idea is to flush a target cache line and measure the time this operation takes to finish. If the execution time of the eviction is bigger than a certain threshold than the target line was refilled by the victim process, otherwise not [18].

S$A This attack try to overcome the main limitation of Flush+Reload, which is the requirement of page deduplication. This attack was ideated by Irazoqui et al. in [5] and it detects accesses to the LLC by using huge size pages to allocate data of the malicious process.

3.2 Flush+Reload Attack

Flush+Reload (FR) is an intrusive cache SCA introduced by Yarom and Falkner in 2014 [49] based on the work of Gullash [19]. It is an inter-process and inter-virtual machine SCA which allows to monitor when a victim process has run one or more target instructions (i.e. to know what it has done). This attack is based on the fact that pages and the last level of cache are shared between programs that use them, as stated in Sections 2.1.1 and 2.2. It

allows to monitor a routine implemented by a shared library interfering with the cache system. The requirements of this attack are that the attacker can run a malicious process on the same machine of the target one and this process can load the target shared library. What makes possible this attack is that the principle of isolation is not respected at hardware level. This idea can be extended in a virtual-machine environment when the host OS uses a technique called content-based page sharing. This optimization makes to Host OS to aggressively scan pages and merge identical ones and it is present both on Linux [6] and Windows [24].

A malicious program can monitor what another program is doing evicting the cache line associated to a target instruction of the shared library (using the *CLFLUSH* instruction on x86, or any other equivalent method) after it has loaded the shared library in its adress space. This means that the attacker program invalidates the line of the caches that contains the monitored instruction. Now the attacker process waits for a fixed amount of time and it tries to access the target instruction. If the time required to execute the access is small (i.e. under a certain threshold) means that the victim process has accessed the instruction (it previously triggered the refill system of caches), otherwise the victim process didn't have executed the instruction. The operation is then repeated.

The core of the Flush+Reload method is Listing 1. Line 13 invalidates *adrs* and then the algorithm wait some time before calling again this function on *adrs.* Line 10 loads the content pointed by *adrs* and lines 7 and 12 collects the time before and after the load operation. The inline assembly block saves the difference between the clocks collected in lines 12 and 7 in *time.* If *time*<*threshold* then the victim process has accessed the instruction. Figure 3.2.1 represents the flow of a FR attack in case the target program accessed the target instruction. The shared page is the one colored with green and all the cache lines associated are colored with the same color. State (a) is the initial condition: the cache line associated to the target instruction is present in all the caches. The malicious process forces the eviction of the target cache line from all the cache system triggering the transition to state (b). After that, the target process must use the target instruction, so it induce a refill

Listing 1 Flush+Reload core function written in C.

```
1  int probe(char *adrs) {
2    volatile unsigned long time;
3    asm __volatile__ (
4      "mfence\n"
5      "lfence\n"
6      "rdtsc\n"
7      "lfence\n"
8      "movl %%eax, %%esi \n"
9      "movl (%1), %%eax \n"
10     "lfence\n"
11     "rdtsc\n"
12     "subl %%esi, %%eax\n"
13     "clflush 0(%1)\n"
14     : "=a" (time) : "c" (adrs) : "%esi", "%edx");
15   return time < threshold;
16 }
```

of the shared L3 and its L2 and L1 caches. This action leads to the transition to state (c). Now the spy process probes the target instruction, inducing the refill of its private L2 and L1 caches. Now the condition is returned back to state (a).

Yarom and Falkner managed to succesfully execute this attack on RSA implementation of *GnuPG 1.4.14* (using *libgcrypt 1.5.3*) in [49] in 2014. In the same year Yarom and Benger managed to recover OpenSSL ECDSA nonces with a FR attack [48]. Genkin et al. broke some applications of Curve25519 based on *libgcrypt [16]*. Bernstein et al. in [10] set up a FR attack with target RSA 1024 and 2048 with CRT optimization of *libgcrypt 1.7.6* using a combination of Flush+Reload, Performance-Degradation Attack ([3]) and a modified version of Prune-and-Branch algorithm of Heninger and Shacham [22]. Note that performance degradation attack uses the same principles of Flush+Reload on target addresses but without probing phase. In this way the target process is slowed down due to cache misses. This system may increase noise but it also allows to perform a Flush+Reload attack (or other attacks that relies on high-frequency samples) in a more relaxed environment. Research also tried to detect or mitigate FR attacks. The

simplest countermeasures are the following:

1. Limit the usage of the *RDTSC* instruction like by setting the *CR4* register [49].

2. Disable of page sharing. This seems to be the most effective countermeasure [5].

Also a system that describes detection of FR attacks with PMCs is described in [12].

3.3 Intel RAPL

Since Performance Monitoring Counters were introduced on Intel architectures a lot of research followed the trend to estimate power consumption with them starting from [13]. The Intel RAPL interface was introduced in 2011 with SandyBrige architecure, and most of the related works are focused on validating such counters, but a lot of the details of this interface remain hidden because the Intel offical documention lacks in the description. Desrochers et al. validated the energy values produced by RAPL in [14], especially DRAM power zone on Haswell architecture while previous works focused on SandyBridge architecture and core power zone [43]. The authors of [14] revealed that DRAM measurements on Haswell architecture are quite accurate (but they may be under-estimated in case of low-operation conditions), while other power zones have an uncertainity of about 20%, but the behaviour of the system is well described. Hahnel validated the update frequency of RAPL that has an imprecision of 50000clk [21] and built an instrumentation system to measure short-code paths.

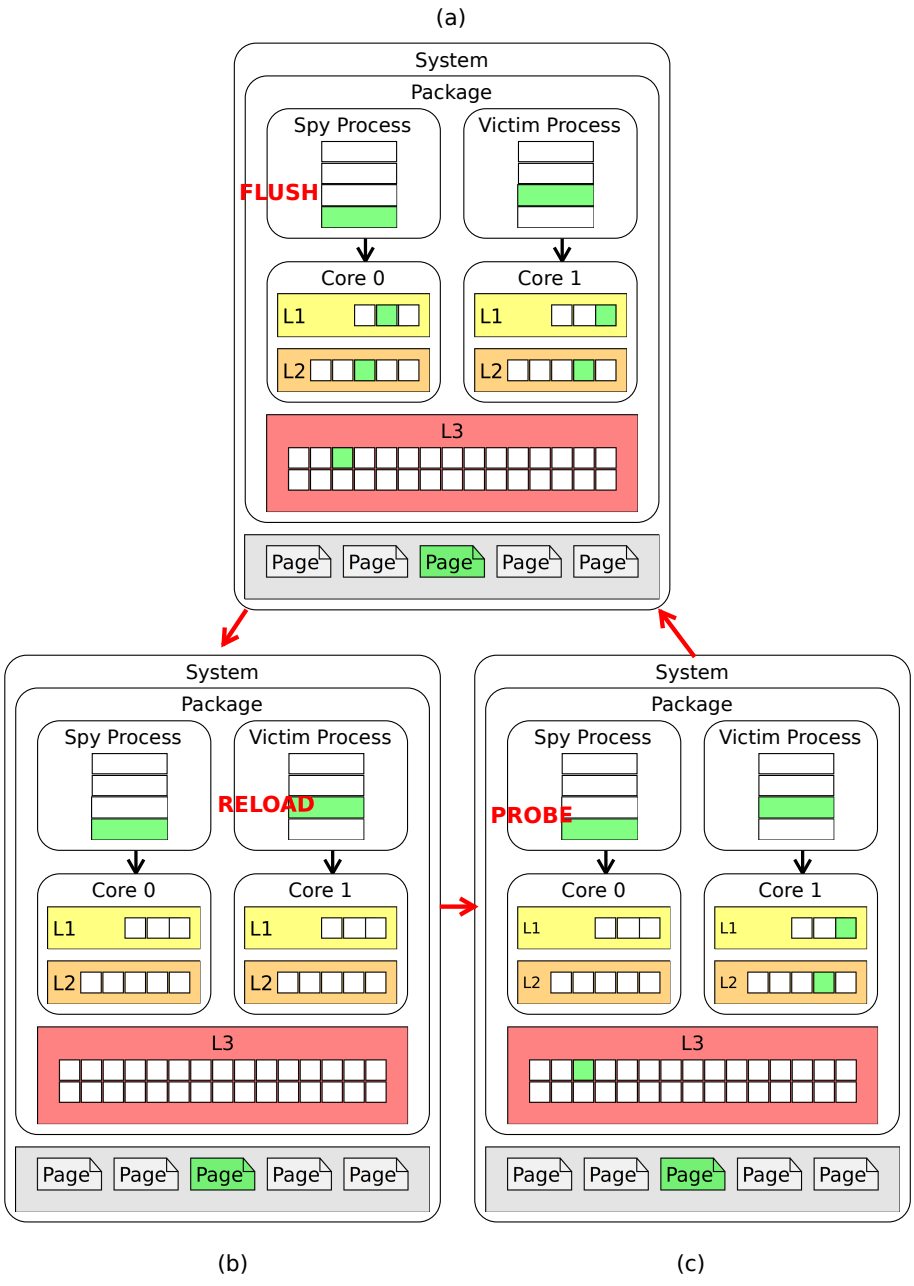

Figure 3.2.1: The working cycle of Flush+Reload Attack in case of an access.

Chapter 4

Proposed Methodology

This chapter describes the design of the proposed methodology that aims to detect if a target hardware performance counter leaks information about the execution of specific operations. Section 4.1 shows how to build a set of test-cases and the statistical analysis required to perform this verification. Section 4.2 shows how the information discovered with the previously cited methodology could be used to leak information from an application

4.1 Information-Leakage Verification Methodology

This methodology requires to select a target hardware counter *hwc*. It is assumed that the access to *hwc* is not restricted. It must be known the update period T_{hwc} and the sensitivity S_{hwc} of the target counter. The idea is to design a set of test-cases E, where every test $e \in E$ tries to "stimulate" the target counter. Every test-case should test a specific *operation* with a specific *flavour*. For example, a test composed by a sequence of multiplications may be labelled with *MUL* as *operation.* If all the operations of the test always multiply by a factor of one, this test may have the *flavour* label set to *ONE*; but if the factor is always zero, then the *flavour* label may be *ZER*. After the collection of the results of the experiment, different Welch's *t-tests* are performed. This statistical test verifies the null hypothesis that

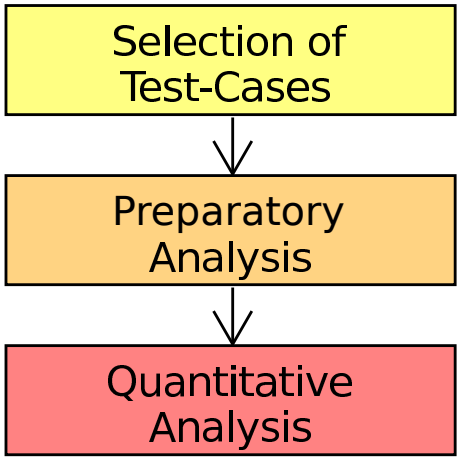

Figure 4.1.1: Steps of the proposed methodology.

two populations with different variances have equal means. The analysis is performed in three steps:

1. Selection of the test-cases.

2. Perform statistical analysis to detect if test-cases with same *operation* but different *flavour* has different mean considering n_{iter} identical.

3. Considering the operations that yeld significant results in the previous point, try to compute a quantitative estimation of the *operation* with specified *flavour*.

4.1.1 Selection of Test-Cases

Every test-case $e \in E$ should produce a result r^e. This result is computed by probing the value of *hwc* before and after the execution of the test case, respectively bef^e and aft^e. So, the the result is computed with a simply subtraction, which means $r^e = aft^e - bef^e$. *operation* and *flavour* are not the only labels associated to every test case. it is better to associated to every test a tuple composed by *(unit,operation, flavour,*n_{iter}*)*. In detail:

unit This is the hardware component that the test should stimulate during its execution. For example, a test-case which performs add should be

labelled with *ALU*. A test-case that test jumps may be labelled with *control-flow* or a test-case which tests cache effects could be labelled with *cache*.

operation It adds an additional detail to the *unit* label. It tells which particular operation is tested. If a test-case uses only add operations it can be labeled with *ADD,* a *control-flow* labelled operation which tests backward branches can be labelled with *JMB* (JuMp Backward) and a *cache* labelled test-cases which implements *RC4* algorithm can use *RC4* as label.

flavour It adds an additional information to what specific aspect of *operation* is tested. If a test-case with *operation* as *ADD* cause an overflow, then its *flavour* may be *OVF* (OVerFlow). A *JMB* operation may test a case where all the branches are set to false, so it has the *flavour* label set to *FAL*. A *RC4* algorithm in which are forced cache misses with *CLFLUSH* instruction may have it *flavour* set to *CLF*.

n_{iter} It tells how many times the *operation* with specific *flavour* is repeated in a single test-case.

The order of the elements in the tuple is not casual, but it indicates the order on how explore the space of all the possible test-cases and selecting the correct ones. At first, considering the target counter *hwc*, the hardware *unit*s that influences *hwc* should be selected. After choosing a set of "influent" units, then it must be possible to identify a set of *operation*s that are executed on them. Every considered operation may have different versions, or different *flavours.* Only combinations of *(unit,operation, flavour)* that influences *hwc* should be considered. The last element to investigate is n_{iter}. This value is used to resolve the temporal and sensitivity constraints introduced by *hwc*.

With this setup, every atomic operation of the test-case with operation label *op* and flavour *flv* will run for a time $t_{op,flv}$, so the execution time of the entire test-case is $t^e_{op,flv} = n_{iter} \cdot t_{op,flv}$. The same reasoning is performed with the impact of the test-case on the target hardware counter value: the atomic operation of the test-case has an impact of $i_{op,flv}$, but there could be

some noise $k \sim WN(\mu, \lambda^2)$ in the system, so the global impact is $r^e_{op,flv} = n_{iter} \cdot i_{op,flv} + k$ .The test-cases must be designed in the way that their execution time is greater or equal then the update period of the target counter ($t^e_{op,flv} \geq T_{hwc}$) and the impact of the test-case must be at least equal to its sensitivity ($r^e_{op,flv} \geq S_{hwc}$).

4.1.2 Preliminary Analysis

This broad-search test aims to detect a possible leak of a target hardware counter in a qualitative way. The leakage is verified by performing Welch's *t-tests*. If it is possible to observe a difference between the impact of two operations with same *operation* label, but different *flavours,* then it is be possible to observe when one of them takes place by simply monitoring the target counter *hwc,* assuming that it is known that a specific *operation* is executing. Considering the set of test cases E designed in the previous section, they must be deisgned with the constraint that test-cases with the same *operation* label must have the same n_{iter}. Every test case is executed N_1times. If *t-tests* associated to an *operation* yeld significant results with α_1, then it is reasonable to indagate it in a more detailed manner with the successive test.

4.1.3 Quantitative Analysis

This test tries to estimate quantitatively the impact of the operations executed in the test-cases on the target counter. Every test-case is run N_2 times and it is identified by a the tuple as previously specified. In this test n_{iter} is not binded to the operation type, but it is a value that belongs to a set that allows to compute the estimation precise enough without occurring in excessive execution time of the test-case. Every test-case should produce the result $I^e_{op,flv} = \frac{r^e_{op,flv}}{n_{iter}}$. If n_{iter} is big enough w.r.t the order of magnitute of k, then:

$$I^e_{op,flv} = \frac{r^e_{op,flv}}{n_{iter}} = \frac{n_{iter} \cdot i_{op,flv} + k}{n_{iter}} = i_{op,flv} + \frac{k}{n_{iter}} \cong i_{op,flv} \qquad (4.1.1)$$

*t-test*s with confidence interval α_2 are used to know if there is a significant difference between test-cases. The tasks to perform in this tests are the following ones:

- By Fixing *operation* and n_{iter}, perform *t-test*s between *populations* with different *flavour*. This tests should yeld significant results (this fact is equal to different means of the two populations). This is also an additional check of the first analysis.

- Fixing *operation* and *flavour, o*bserve if the tests with different n_{iter} produce similar results. If the tests have been designed correctly, there should be a convergence to the real value, thanks to the Equation (4.2.1).

The result of this verification is an estimate of the impact $\widetilde{i}_{op,flv}$ of a singular operation *op* with specifed flavour flv on *hwc*.

4.2 Methodology for Information-Leakage Analysis of an Application

After the execution of the proposed methodology discussed in the previous section, a set of operations that can be distinguished by knowing the associated *flavour* is available. Also an estimation of the impact $\widetilde{i}_{op,flv}$ with *operation op* with *flavour* flv is known. It is possible to use the knowledge of these values to extrapolate information about target program P with input k by looking at the target counter *hwc*. A trace tr is an ordered sequence of sampled values of *hwc* at a fixed frequency f_{tr}. The assumptions are the following ones:

- All the implementation details of P are known.

- The execution path (the sequence of executed instructions) of P depends uniquely on the input k or it may be fixed. This means that if k is known, then the execution path is known, otherwise not. If the

path is fixed, since we know all the implementation details, then it is known.

- It is not possible to induce any fault or crash of P with perturbation. The program always terminates and it always executes the same path of execution giving the same input, but is possible to perturbate the behaviour of P in some way forcing it to change its execution. The idea is to replace the execution of the *operation* of *flavour* i with *flavour* j by running a malicious process.

The test is composed by the following steps:

1. **Code Analysis**. Since the implementation details are known, an analysis of the code of the target program must be performed to know as good as possible the target instructions suitable for achieve the purpose.

2. **Comparative Analysis**. Trace the target counter while running a target program in two case: with and without the disturbance. The result of this step are two traces tr_{base} and tr_{dist}. They are respectively the trace without and with the disturbance. By coparing the two traces it is possible to know where the replacement took place.

The step 2 may be repeated a variable number of times and the inputs may be change. The number and modes of repetition are case-study dependant and they cannot be decided in this context.

The key point is to choose the correct operation to be perturbated. By choosing the right operation and comparing the two traces it is possible to observe when a data-dependent operation is performed. A toy example is shown in Figure 4.2.1. The blue trace is the result of monitoring the target counter without perturbation and the green trace is the one with perturbation. The red points signal the target instruction that is perturbated. The result is an increment of the impact of the target instruction on the target counter. The case-study in Chapter 6 will focus this analysis on a cryptographic routine by forcing a cache miss in a FR fashion. The idea is to induce a cache miss and observe it by looking at Intel RAPL counters.

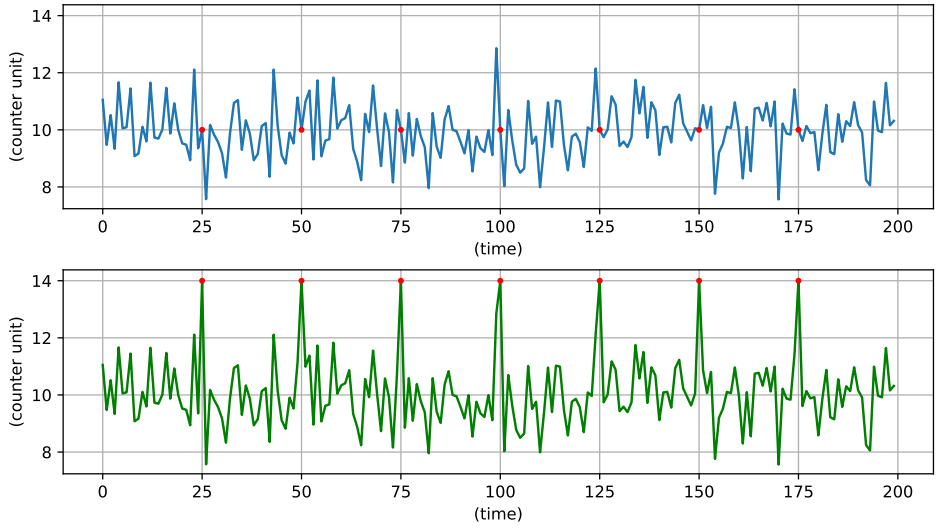

Figure 4.2.1: A toy example of perturbation of a target program P.

Chapter 5

Implementation

This chapter displays the tools used to implement the proposed methodology. All the tools used are related to the collection of the values related to hardware performance events. The employment of these tools will be shown in the Chapter 6.

5.1 Tools

Since hardware performance monitoring requires kernel code to program MSR, the software must use system calls to access performace counters and limit the possibilty of attacker of reading sensitive information. To make the access to performance monitoring simple, a lot of software solutions have been developed. PAPI and Perf are the most common solutions for performance monitoring.

5.1.1 Powercap and PAPI

The *Powercap* framework is a consistent interface for reading and writing RAPL counters from user-space[7] in Linux kernels. It can be enabled by loading the module *intel-rapl*. Without any particular privilege it is possible to read the consumed energy in μJ of different zones of the package. Root privileges can also set a cap to the power consumption of the core. it is possible to access the *Powercap* interface in the system folder

```
/sys/devices/virtual/powercap/intel-rapl
    - intel-rapl:0/
    - name = package-0
    - energy_uj
    - intel-rapl:0:0/
      - name = core
      - energy_uj
    - intel-rapl:0:1/
      - name = uncore
      - energy_uj
    - intel-rapl:0:2/
      - name = dram
      - energy_uj
```

Figure 5.1.1: Structure of the Linux Powercap interface.

/sys/class/powercap/intel-rapl. In this folder *package-*x** is mapped with the directory *intel-rapl:x*. In every directory named *intel-rapl:x* there are the interfaces under the form of files and the subzones are mapped as *intel-rapl:x:y*. Every directory following the regex *intel-rapl[:n]+* has a file named *energy_uj* which contains the energy consumption in μJ associated to that zone. The file *name* gives the name to the power zone. Names identify the power zones listed in Section 2.1.2. As we'll be shown in the case study, the fact that the reading on RAPL counters through this interface can be a side-channel.

PAPI *[2]* is a *C* library that allows to instrument code for performance monitoring using PMCs. It also tries to leverage the differences between different architectures by providing a uniform interface. The usage of PAPI allows the programmer to write code which executes performance monitoring. PAPI is also easily extendible programming user-defined components. In fact, there's a component in the default distribution that allows to monitor RAPL counters through the Powercap interface. This components simply memory-maps the files of Powercap and allows to access them just like any performance counter in PAPI.

5.1.2 Perf

Perf is a performance-monitoring suite available on Unix systems and it is integrated in the linux-kernel. Just like PAPI, it allows to monitor *PMC*s. It can be used both for counting or tracing of PMCs and it also offers an easy command-line interface. *Perf* will be used to profile the sign routine to know some basic information like the execution time of a modular multiplication operation. *Perf* usage is controlled by a system option called *perf_event_paranoid* at path */proc/sys/kernel/perf_event_paranoid* that can be changed by the system administrator. The possible levels of *perf_event_paranoid* are the following ones:

-1 Everything is permitted

0 Disallow raw trace point access for non-root

1 Disallow cpu events for non-root

2 Disallow kernel profiling for non-root

3 Disallow anything

The lowest is the value, more power has a non-privileged user. A high level *perf_event_paranoid* implies all the restriction of lower levels.

5.2 Energy Tracer

This is a tracer of the consumed energy of the whole system built with PAPI from scratch. How the designed tracer interacts with PAPI and consequently with the hardware is shown in Figure 5.2.1. The energy tracer calls PAPI with the function *PAPI_read* and PAPI simply read the value offered by Powercap. Powercap updates its values using kernel code to read *MSRs* associated to Intel RAPL interface with *RDMSR*.

The sample period of the energy tracer is $\frac{1}{3}ms$. This sample period give enough time to the PAPI library to read RAPL counters of all the power zones between one sample and the successive and the signal can be fully

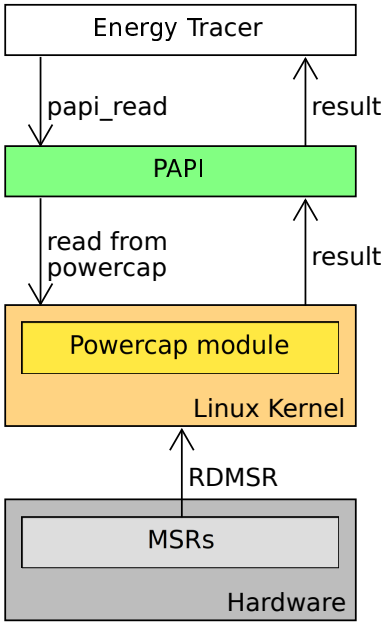

Figure 5.2.1: How the energy tracer interacts with PAPI and consequently with the kernel and the hardware.

reconstructed. The energy readings are presented to the user in a CSV format. This output can be used for offline analysis. The tracer has also the capability of evicting a set of specified instructions associated to a shared library from the memory system using *CLFLUSH* exploiting the vulnerability of page sharing on which FR attacks rely on. The cache eviction task is performed on a different thread spawned with the *pthread* library.

The offset of an instruction in a shared library can be easily fetched inspecting the target shared object with *GDB or objdump*[1]. The portion of code responsible of evicting instructions from the caches is inspired by the Mastik [2][47] implementation of Flush+Reload. The shared library is loaded in the process space of the tracer via *mmap* function. Virtual addresses in the tracer address space are fetched with the same function. The fetch of the virtual address must be performed because inspecting the assembly code

[1] *objdump* is a Linux command line tool used to explore object files. This utility will be used to disassemble the shared library obtained from the compilation of *libgcrypt*. *libgcrypt* is compiled by default with debug symbols, so it is very easy to explore disassembled code.

[2] Mastik is a toolkit that provides C implementations of cache side-channel attacks.

of a shared library just shows offsets from the begin of the area of memory that will contain the shared library. When a shared library is loaded during the start of a process it is put in an area of memory that changes at every run due to ASLR. The Global Offset Table and the Procedure Linkage Table allows to acces the functions offered by the shared library[30].

The tracer can run in two modes:

no_evict Just monitor energy consumption

evict Monitor energy consumption and evict addresses. In this case an additional thread is instantiated which is responsible of the eviction as described above

This tracer will be used as malicious process in the case study to monitor RAPL counters. Figure 5.2.2 describes the execution flow of the energy tracer. If the eviction mode is active (*evict_mode* set true), then a thread responsible of performing all the evictions is spawned. Since the time between one sample and another is low.

The precision if this tool is mainly lmited by the resolution of the target hardware counter. Since the experimental results will read values from the Intel RAPL counters (the update period of $976\mu s$ and the vertical resolution of $61\mu J$, as will be shown in Section 6.1), the results obtained by such traces will suffer of low resolution of the raw values. This fact is a great limitation if an attacker wants to perform a SCA based on these counters. For example, a FR attack has a resolution of thousand of cycles, which is greatly below the resolution of RAPL.

Moreover, the energy tracer is a software that runs on an processor on which a lot of other processes is running. This chaotic environment introduces unpredictabilities in the execution of the energy tracer (and also on the sign routine), so the sample frequency has variations between one sample and another. Also the noise in the system is not negligible.

For all these reasons, the case study won't focus on the analysis of one single trace, but it will collect and average a reasonable amount of them and perform statistical operations when needed.

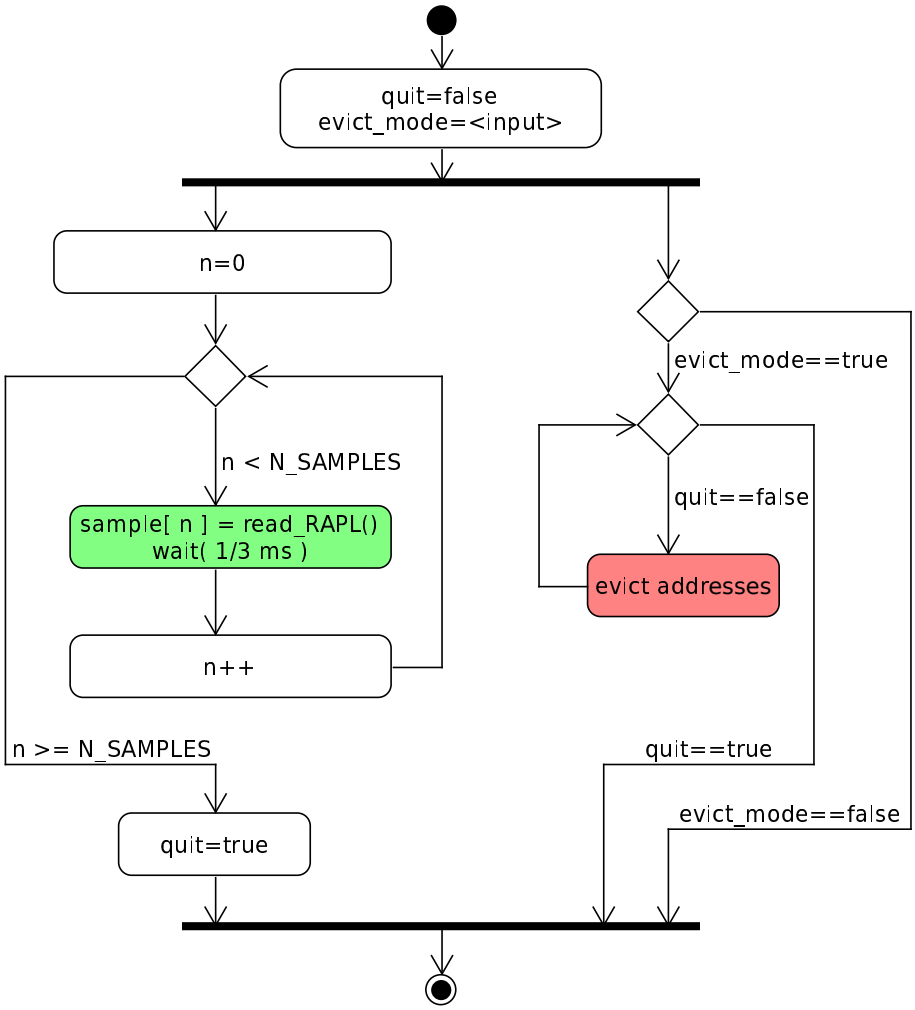

Figure 5.2.2: Activity diagram of the energy tracer.

Chapter 6

Experimental Results

This section presents the results of the information-leakage analysis performed on the target architecture considering as target counter all the Intel RAPL counters, except the off-core one. Finally, a case study is proposed. Before presenting the results, a description of the setup of target hardware and software is offered to the reader. Section 6.1, in fact, describes in detail the target hardware and software, while Section 6.2 reports the results of the implementation of the proposed methodology discussed in Chapter 4. Section 6.3 reports the results of the case-study on RSA-16384 by using the methodolgy presented in Section 4.2.

6.1 Experimental Setup

The tests were run on an ASUS F-302L, which is a everyday-use laptop. The installed processor is the Broadwell-U processor running at the fixed frequency of $800MHz$. Broadwell is the 5th generation of Intel Core and it is an adaption of the Haswell architecture with a $14\eta m$ processor instead of a $22\eta m$ one. This architecure follows the tendency of Intel starting from Nehalem of using 3 levels of cache using the following logic: every core has its own L1 and L2 caches and the L3 is shared by all the cores on the package. Note that L3 cache is inclusive, which means that all the data contained in L1 or L2 is surely contained in L3 [27], but the reverse is not necessary true.

The L1 cache is divided in two different caches, one for instructions and one for data to increase the bandwidth. Technical details of the caches are the following ones [28]:

L1 (both Data and Instruction) 32 KiB 8-way set associative, 64 B line size with write-back policy

L2 256 KiB 8-way set associative, 64 B line size with write-back policy

L3 3 MiB, 64 B line size with 12-way set associative with write-back policy

it is easy to observe that the line size is preserved among all the cache levels.

In this architecture it is possible to monitor at most 4 performance counters simultaneusly per process (8 without hyper-threading technology). RAPL controls and monitors the energy consumption of the package, the aggreated energy consumption of all the cores, the DRAM and an off-core component. In this architecture the off-core component is the GPU, which is not relevant for this study. These counters are updated at the frequency of $976\mu s$. Sensibility of RAPL counters of every power zones is $61\mu J$.The hypothesis is that this architecture has (FIVR) because Haswell has [20], so it is reasonable to assume that a voltage probing approach is used instead of a modeling one. Samples also demonstrate that the granularity of the energy counters is $61\mu J$ for all the power zones.

The DRAM memory is a SK Hynix HMT451S6BFR8A of 4GB.

The target operating system is Linux 4.12.8-2 and it implements the page-sharing optimization. The energy tracer, *libgcrypt* and PAPI were compiled with *GCC 7.2.0*. The version of PAPI is 5.5.1.0. The version of *libgcrypt* is *1.7.6*. *perf* ran with version *4.13.g569dbb*.

RAPL counters can be read from user-space without privileges. This can be done if Linux has the *Powercap* module enabled. In this study the level of *perf_event_paranoid* is set to 2, which means that it allows the monitoring of performance counters in user mode and only related to a specified process.

In conclusion, this chapter describes the test and the case study relying on the fact that they were run on the described architecture with the specified software.

6.2 Characterization of RAPL Data Dependence

This test verifies the information-leakage of the Intel RAPL counters by performing a set of tests as described in the Chapter 4. The tests were written in assembly (using *NASM[38]* or embedded C-assembly), while all the code responsible collecting energy values of RAPL counter via PAPI is written in C. The target RAPL counters are package, cores and DRAM.

6.2.1 Selection of the Test-Cases

The selected units are the *ALU*, the *control flow* and the *Cache*. All the chosen units surely influences the package and core power consumption because all the instructions are executed in cores, and the cores are contained in the package. The DRAM is tightly coupled to the core execution and it may exposes information about *control-flow* and *Cache*, since non in-cache instructions must be fetched from the main memory. Arithmethic and logic operations are perfomed on registers, so there are no expectations in finding a significant results in DRAM power zone for this type of operations. *control-flow* operations test the impact of jumps on power consumption.

ALU operations tests both basic arithmetic (such as add, multiply and division) and logic (and, or, shift) operations, *control-flow* tests backward and forward jumps in the condition they're always true or false. *Cache* operations uses a RC4 algorithm keeping elements in cache or out-of-cache (via cache eviction). The designed tests are displayed in Table 6.1.

Table 6.1: The test-cases chosen during the selection step in Section 6.2.1.

unit	**operation**	**flavour**	$\mathbf{n_{iter}}$	**Description**
Cache	RC4	INC	$2.56 \cdot 10^6$	RC4 in cache
Cache	RC4	CLF	$2.56 \cdot 10^6$	RC4 flushing cache
Flow	JMP	TRU	10^8	Forward branch true
Flow	JMP	FAL	10^8	Forward branch false
Flow	JMB	TRU	10^8	Backward branch true
Flow	JMB	FAL	10^8	Backward branch false
ALU	ADD	SMP	$130.21 \cdot 10^6$	Simple add
ALU	ADD	CAR	$130.21 \cdot 10^6$	Test carry flag
ALU	ADD	OVF	$130.21 \cdot 10^6$	Test overflow flag
ALU	NOP	SMP	$130.21 \cdot 10^6$	Test nops
ALU	AND	ZER	$130.21 \cdot 10^6$	$n \,\&\, 0$
ALU	AND	AON	$130.21 \cdot 10^6$	$n \,\&\, (2^{32} - 1)$
ALU	AND	SEL	$130.21 \cdot 10^6$	$n \,\&\, n$
ALU	MUL	SMP	$130.21 \cdot 10^6$	Generic multiplication
ALU	MUL	TWO	$130.21 \cdot 10^6$	$n * 2$
ALU	MUL	ZER	$130.21 \cdot 10^6$	$n * 2$
ALU	MUL	DZE	$130.21 \cdot 10^6$	$0 * 0$
ALU	MUL	ISM	$130.21 \cdot 10^6$	Signed multiplication
ALU	DIV	SMP	$130.21 \cdot 10^6$	Generic division
ALU	DIV	TWO	$130.21 \cdot 10^6$	$n/2$
ALU	DIV	ZER	$130.21 \cdot 10^6$	$0/n$
ALU	DIV	ONE	$130.21 \cdot 10^6$	$n/1$
ALU	OR	ZER	$130.21 \cdot 10^6$	$n \,\vert\, 0$
ALU	OR	AON	$130.21 \cdot 10^6$	$n \,\vert\, (2^{32} - 1)$
ALU	OR	SEL	$130.21 \cdot 10^6$	$n \,\vert\, n$
ALU	SHR	AOO	$130.21 \cdot 10^6$	R-shift $2^{32} - 1 >> 1$
ALU	SHR	AOT	$130.21 \cdot 10^6$	$2^{32} - 1 >> 2$
ALU	SHR	AOS	$130.21 \cdot 10^6$	$2^{32} - 1 >> 16$
ALU	SHR	AOM	$130.21 \cdot 10^6$	$2^{32} - 1 >> 32$

Table 6.2: Result of the *t-tests* of the preliminary analysis with $\alpha_1 = 0.001$ between populations of *control-flow* and *Cache* unit with the same *operation.*

Operation	**Flavour 1**	**Flavour 2**	**p-val pkg**	**p-val cores**	**p-val DRAM**
JMB	FAL	TRU	$5.03 \cdot 10^{-14}$	$2.20 \cdot 10^{-16}$	$4.44 \cdot 10^{-16}$
JMP	FAL	TRU	$6.55 \cdot 10^{-15}$	$1.18 \cdot 10^{-15}$	$4.51 \cdot 10^{-14}$
RC4	CLF	INC	$3.35 \cdot 10^{-16}$	0.25	$2.82 \cdot 10^{-15}$

6.2.2 Preliminary Analysis

This preliminary test aimed to detect if RAPL counters leak some basic information using the test-cases described in the previous section. Test functions were written in assembly, compiled with *NASM* and linked together with a main written in C. The main function is responsible of initializing PAPI (whom is used as interface with RAPL counters through *Powercap*) and collecting energy measurements. Every test-case was performed $N_1 = 500$ times. Simple Welch's *t-test*s on two populations with different variance and $\alpha_1 = 0.001$ were performed among group of test-cases with the same *operation* and considering the same power zone.

The results are shown in Tables 6.2 and 6.3. There are significant results in case of the control flow: it is possible distinguish if a branch is taken or not taken knowing if it was a backward or a forward branch. *t-test*s related to *JMP* and *JMB* operations demonstrate this hypothesis. It is also possible to distinguish case where there are high cache accesses w.r.t cases with absent cache accesses. *RC4* tests were, in fact, focused on demonstrating this fact. Obviously, having data in cache consumes less than fetching it from main memory. It is important to observe that no one of the operations of the *ALU* unit yelded significant results.

Table 6.3: Result of the *t-tests* of the preliminary analysis with $\alpha_1 = 0.001$ between populations of *ALU unit* with the same *operation*.

Operation	**Flavour 1**	**Flavour 2**	**p-val pkg**	**p-val cores**	**p-val DRAM**
ADD	CAR	OVF	0.64	0.79	0.38
ADD	CAR	SMP	0.63	0.98	0.26
ADD	OVF	SMP	0.99	0.80	0.79
AND	AON	SEL	0.69	0.65	0.62
AND	AON	ZER	0.92	0.91	0.80
AND	SEL	ZER	0.62	0.57	0.47
DIV	ONE	SMP	0.91	0.94	0.32
DIV	ONE	TWO	0.81	0.81	0.45
DIV	ONE	ZER	0.69	0.69	0.92
DIV	SMP	TWO	0.88	0.86	0.80
DIV	SMP	ZER	0.61	0.64	0.37
DIV	TWO	ZER	0.51	0.52	0.51
MUL	DZE	ISM	0.95	0.95	0.39
MUL	DZE	ONE	0.93	0.99	0.50
MUL	DZE	SMP	0.73	0.84	0.44
MUL	DZE	TWO	0.74	0.75	0.13
MUL	DZE	ZER	0.90	0.90	0.22
MUL	ISM	ONE	0.98	0.95	0.85
MUL	ISM	SMP	0.68	0.89	0.95
MUL	ISM	TWO	0.69	0.80	0.50
MUL	ISM	ZER	0.95	0.95	0.71
MUL	ONE	SMP	0.66	0.85	0.90
MUL	ONE	TWO	0.67	0.76	0.40
MUL	ONE	ZER	0.96	0.91	0.58
MUL	SMP	TWO	0.99	0.91	0.48
MUL	SMP	ZER	0.64	0.93	0.67
MUL	TWO	ZER	0.64	0.84	0.76
ORL	AON	ZER	0.87	0.89	0.78
SHR	AOM	AOO	0.75	0.50	0.56
SHR	AOM	AOS	0.18	0.10	0.04
SHR	AOM	AOT	0.40	0.26	0.20
SHR	AOO	AOS	0.29	0.31	0.17
SHR	AOO	AOT	0.59	0.63	0.49
SHR	AOS	AOT	0.60	0.59	0.48

Table 6.4: Test-cases of the quantitative analysis.

unit	operation	flavour	$\mathbf{n_{size}}$	Description
ICache	LOA	HIT	$\forall n \in P_{iter}$	Load an in-cache instruction
ICache	LOA	MISS	$\forall n \in P_{iter}$	Load a non in-cache instruction

6.2.3 Quantitative Analysis

The previous test revealed that RAPL counters leak information about cache hits or misses, but the previous test considered only data cache operations. In this test the focus is on instruction cache, so the previous tests are slightly modified to test instruction cache hits and misses. The goal of this test is to compute both the energy consumption of a cache instruction hit and of a cache instruction miss and also their latency. Foreach flavour, a instance of test-case is run with $n_{iter} = x | x \in P_{iter}$. $P_{iter} = \{10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8\}$ is the set that contains all the possible values that can be assigned to n_{iter} in this test. The performed test-cases are shown in Table 6.4 .

The eviction or the presence of the instruction in cache (before other instructions are executed) is ensured by introducing *MFENCE* and *CPUID* after the loading or flushing the instrucion from cache by using a simple load for the hit case or *CLFLUSH* for the miss case. The *MFENCE* instruction is a barrier for the memory operation: Every load and store instruction that precedes in program order the MFENCE instruction is globally visible before any load or store instruction that follows the *MFENCE* instruction is globally visible[28]. This also means that the execution is performed out-of-order, but the effects are ordered in the ROB. *CPUID* also ensures serialization of instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed[28]. The two tests are both introduced and concluded by PAPI functions to collect energy reading of all the available power zones. With a high number of iterations (n_{iter}) all the border effects should be nullified, but the latency introduced by the serializing instruction (*CPUID*) must be subtracted by the result.

Table 6.5: Energy consumption results of the quantitative analyis on Intel RAPL counters considering instruction cache hit and miss.

zone	**flavour**	**estimate [μJ]**
package	hit	0.84
	miss	1.40
core	hit	0.20
	miss	0.33
DRAM	hit	0.13
	miss	0.26

Every test-case was repeated $N_2 = 500$ times. *t-test*s were performed with $\alpha_2 = 0.001$ to verify if there was a significant difference between a hit or a miss considering the same power zone and same n_{iter}. It was possible to observe significant results (i.e. different means between hits and misses) with $n_{iter} \geq 10^3$. With the same value of n_{iter}it was possible to observe the convergence to the real value of energy consumptions of the tested instructions. The energy estimations of a single cache hit or miss foreach power zone are displayed in Table 6.5.

Every test-case contains a *CPUID* instruction that introduces a time penalty because it is slow. This factor must be subtracted to the results of the latency. So an instruction cache hit has a cost of $\tilde{t}_{hit} = 786clk - 280clk = 506clk$ and an instruction cache miss has a cost of $\tilde{t}_{miss} = 1250clk - 280clk = 970clk$.

6.3 Case Study: Information Recovery from RSA-16384

Considering the discussion of Section 4.2, we want to replace instruction cache hits with instruction cache misses of lines associated to a set of target instructions. From the previous analysis, it is known that an instruction cache miss consumes more energy than a cache hit. If it is possible to force a cache miss in the target process, then this event is observable in the energy trace. The perturbation is demonstrated to be possible thanks to the FR dissertation: *CLFLUSH* can be used by a malicious process to induce a cache miss. The proposed experiment combines SPA and FR using RAPL counters and it tries to recover all the possible information about the signing routine of RSA-16384 implementation of *libgcrypt 1.7.6* without both CRT optimization and blinding. The target version this library is demonstrated to be vulnerable against FR attacks [10], so there are clues that if it is possible to evict the correct instructions in the modular exponentiation routine of *libgcrypt* from the cache system and force the system to consume a reasonable amount of power, then it is possible to know when a target instruction is executed by simply looking at the energy trace of the RAPL counters.

The traces of the RAPL counters are collected with the energy tracer described in Chapter 5. This tracer will act like as a malicious process with cache eviction capabilities. From the previous experimental results, the focus is on the DRAM power zone, since it allows to detect easily between an instruction cache hit or miss. The execution time of a *CLFLUSH* instruction is estimated to be around $1000clk$ at the selected processor frequency. The discussed case study can be seen as a preliminary analysis before the execution of a potential SCA.

The limits that does not allow to perform a case study without any particular problem come mainly from the limitations imposed by RAPL counters (update frequency and low sensitivy) and the imprecision of the tracer tool as described in Section 5.2. The high sample period influences the study by leading it to focus on keys with the key size as big as possible (i.e 16384 bits) in order to cause slower modular multiplications w.r.t keys with a lower

number of bits. Also the disabling of CRT optmization was justified by this reason. The update period is fixed w.r.t the frequency, so also the choice of fixing the CPU to the lowest possible frequency was influenced by this reason. The startup of the program may not come at the same moment at every run, the execution of the sign routine may terminate in different time due to the concurrency between processes on the same system and the energy tracer may sample unprecise. These facts make difficult to rely only on the analysis of a simple raw trace. This is the reason why a single test is run N_{cs} times and the resulted trace is produced by averaging readings that belongs to the same $i-th$ sample.

The case-study was performed assuming the following facts:

1. Page sharing to induce cache misses is enabled, just like FR attacks. FR attack can also be extended in cross-VM environment, but this work focuses on a cross-core attack.

2. It is possible to run any process without any-privilege (non-root) on the machine where the cryptographic routine is executed. The described energy tracer will act as the attacker process. It can both monitor power consumption and evict lines. It is also possible to run the target sign routine with known keys.

6.3.1 RSA Background

RSA is probably the most famous asymmetric-key encryption algorithm and it is used in every-day-use applications such as *SSH*. The construction of the RSA keys follows these steps [37]:

1. Choose p and q

2. Compute the public modulus $N = pq$ and the secret $\phi = (p-1)(n-1)$

3. Choose the public exponent e coprime w.r.t ϕ (i.e. $GCD(e, \phi) = 1$) such that $1 < e < \phi$.

4. Choose the private exponent d such that $de \equiv 1 \, (mod \, \phi)$

5. The public key is the couple (N, e) and the private key is (N, d)

Alice can send to Bob a cipher of the message m computing $c = m^e \, mod \, N$ using Bob's public key and only Bob can read the message using his private key computing $c^d \, mod \, N = m$. The owner of the private key can also sign a message m computing $s = m^d \, mod \, N$ and everyone can verify the correctness of a message computing $s^e \, mod \, N = m$. RSA is resistent to brute-force attacks because it is based on the fact that factorize a big number cannot be performed in a reasonable amount of time, so the strength of this algorithm is based on choosing p and q big enough to make the factorization operation non tractable. Doubling the key size slows down cryptographic operations of about 6-7 times on modern systems [29].

A way to speed up the modular exponentiation computation is to introduce the called CRT optimization. It reduces the computational complexity computing $s_p = m^{d_p} mod \, p$ and $s_q = m^{d_q} \, mod \, q$ and combining them executing $h = u * (s_q - s_p) \, mod \, q$ and $c = s_p + h * p$ instead of computing directly $s = m^d \, mod \, N$. The computations of s_p and s_q work with half-size operands and have half-length exponents, leading to a speedup of a factor 2-4 [10]. $d_p = d \, mod \, (p-1)$, $d_q = q \, mod \, (q-1)$ and $u = \frac{1}{p} \, mod \, q$ are usually stored with the key instead of computing them on the fly to save time.

An option to increase security is to use blinding[11]: instead of computing $s = m^d \, mod \, N$ the algorithm computes $s' = (mr^e) \, mod \, N$ with r a random number such that $GCD(r, N) = 1$ and $1 < r < N - 1$, and then produce the blinded signature with $s'' = (s')^d \, mod \, N$. The correct sign is obtained with $s = (s''rI) \, mod \, N$ where rI is the modular multiplicative inverse of r w.r.t. N. In case of sign, blinding increase resistance to SCAs by obfuscating the original message, but this techinque has been violated [46].

6.3.1.1 Modular Exponentiation with Sliding-window Square and Multiply

Encryption and decryption of RSA-forged messages rely on modular exponentiation, which is the operation that computes $x^y \, mod \, N$. One of the most used method to compute this operation is the square-and multiply method.

All the square-and-multiply methods for modular exponentiation scan the bits of the private key and executes modular multiplications. The number of modular multiplications and what these methods multiply at each step depends on the implementation. The simplest possible implementation of square-and-multiply is the one listed in Algorithm 2 and it is called binary left-to-right modular multiplication, but it is not practically used. In real

Algorithm 2 Left-to-Right binary exponentiation.

Input: Three integers m,d and N, where $d_n...d_1$ is the binary representation of d

Output: $a \equiv m^d \, mod \, N$

```
1: function BINARY_MOD_EXP(m, d, N)
2:     a ← 1
3:     for i ← n to 1 do
4:         a ← a^2 mod N                          ▷ Square
5:         if d_i ≠ 0 then
6:             a ← a · m mod N                    ▷ Multiply
7:         end if
8:     end for
9:     return a
10: end function
```

world applications the sliding window square-and-multiply is employed . This technique both reduces the number of modular multiplication operations and also reduces the operations of precomputation w.r.t other implementations which requires a preliminary step [37]. After selecting a fixed window length W, this algorithm precomputes small odd powers of the base up to 2^{W-1} and reuses them in the exponentiation routine. Algorithm 3 is a pseudo-code implementation of this modular exponentiation method.

The exponentiation routine follows this idea: scan the key in binary form from left-to-right (from MSB to LSB) and perform a square if the bit is set to zero. The exponent is indexed in the way that the MSB is at position n and the LSB is at position 1, assuming that the exponent is composed by n bits. The scan of this number happens from MSB to LSB. If the scanned bit at position i is set to 1 then select the next successive bit set to 1 far k such that $k \leq W$. Now perform a multiplication using b^u where u is the number

Algorithm 3 Left-to-Right sliding window modular exponentiation with sliding window W.

Input: Three integers m,d and N, where $d_n...d_1$ is the binary representation of d

Output: $a \equiv m^d \, mod \, N$

```
1: function LEFT_TO_RIGHT_MOD_EXP(m, d, N)
2:     a ← 1, b_1 ← b, b_2 ← b^2, z ← 0
3:     for i ← 1 to 2^{W-1} − 1 do
4:         b_{2i+1} ← b_{2i−1} · b_2 mod N   ▷ Precompute table of small odd powers of b
5:     end for
6:     i ← n
7:     while i ≠ 1 do
8:         i ← i − z
9:         z ← z+ COUNT_LEADING_ZEROES(d_i...d_1)
10:        l ← min(i, W)
11:        u ← d_i...d_{i−l+1}
12:        t ← COUNT_TRAILING_ZEROES(u)
13:        u ← SHIFT_RIGHT(u)
14:        for j ← 1 to z + l − t do
15:            a ← a^2 mod N                 ▷ Square
16:        end for
17:        a ← a · b_u mod N                 ▷ Multiply
18:        i ← i − l
19:        z ← t
20:    end while
21:    return a
22: end function
```

Step	Bits	Operation	Result
0	\	\	$b^0 = 1$
1	$101_2 = 5_{10}$	M	$b^{0+5} = b^5$
2	0	S	$(b^5)^2 = b^{10}$
3	0	S	$(b^{10})^2 = b^{20}$
4a	\	S	$(b^{20})^2 = b^{40}$
4b	\	S	$(b^{40})^2 = b^{80}$
4c	\	S	$(b^{80}) = b^{160}$
4	$111_2 = 7_{10}$	M	$b^{160+7} = b^{167}$
5	0	S	$(b^{167})^2 = b^{334}$
6a	\	S	$(b^{334})^2 = b^{668}$
6	1	M	$b^{668+1} = b^{669}$

Figure 6.3.1: Left-to-Right sliding Windows exponentiation example with base b and exponent $e = 669$.

identified by the binary representation of $u = (d_i...d_{i-k})_2$ and perform $k-1$ square operations, then restart the scan from the bit after $i-k$. Note that the squares before the first multiply are truncated (squaring 1 results into 1). See Figure 6.3.1 for an example. This example computes the modular exponentiaion of a generic base b for the exponent $e = 669_{10} = 1010011101_2$ with window size $W = 3$. M indicates a multiply, a blue S indicates a square and a magenta S indicates a square introduced by the window (the $k-1$ squares).

The critical issue of exponentiation algorithms with square and multiply is that they leak information about the secret key because the execution path of these kind of algorithms is key dependant. In fact, a lot of SCAs use this vulnerability to recover the secret key.

6.3.1.2 *libgcrypt 1.7.6*

Libgcrypt is the C library that implements the cryptographic operations performed by GnuPG. It is widely used on Unix systems to sign and encrypt documents. GPG is an open-source implementation of the OpenPGP standard. Libgcrypt implements different cryptographic operations: symmetric and asymmetric cryptography, hashing and message authentication

Number	$10\,11\,01\,10_2$			
Limbs	10	11	01	10
Index	3	2	1	0

Figure 6.3.2: A small example of how a number is stored in an MPI structure. In this example every limbs has a size of 2 bits.

code. This document focuses on the RSA implementation of *libgcrypt*. RSA encryption and decryption primitives are implemented with modular exponentation like stated before, so all these operations are built on the call of one function using different arguments.

Since Cryptographic routines usually employes numbers that does not fit a single word of the machine, they use particular implementation of numebr representations. *libgcrypt* employes MPI, which stands for Multiple Precision Integer. MPI takes inspiration from GMP library [33]. A set of ordered Limbs composes an MPI number. They are a division of the bits of a number aimed to fit a word in the machine where the library is executed. In this study limbs have the size of 64 bits. It is possible to see a number implemented with MPI as an array, where every array element is named limb and every one of these limbs stores 64 adjacent bits in a little-endianess fashion [34].

6.3.2 Case-Study Results

6.3.2.1 Code Analysis

This step is necessary to know the details of the implementation and it is the implementation related to Code-Analysis described in Section 4.2. The target code is the modular exponentiation of *libgcrypt 1.7.6*. This step implies both code inspection and profiling the code execution. The eviction point chosen with the code inspection must be selected carefully in the way it is possible to trigger a reasonable increment of the energy consumption. Since the energy consumed by a cache miss is very low w.r.t the sensitivity, it is important to select an instruction that it is executed within a loop a good amount of times. The profiling step allows to find the execution time of the modular multiplication.

The function responsible of performing the modular exponentiation is called *_gcry_mpi_powm* and it is targeted for internal use. It is a C adaptation of the Algorithm 3. The modular multiplication is implemented in the function *mul_mod*. The main difference w.r.t the cited pseudo-code and the real implementation is that *libgcrypt* iterates over limbs of an MPI structure that contains the private key instead of the singular bits using an infinite loop. When all the limbs are scanned, the loop can be interrupted, but there can still be some computation to perform yet. These are the conditions that causes the loop termination:

- While counting zeroes, there are not remaining limbs to iterate over. In this case there will be only square operations out of the loop. This implies that the exponent of the modular multiplication is even. it is like breaking the loop of the algorithm 3 after Line 9.

- Cannot create u of size W because there are not remaining limbs. Instead of computing min in Line 18 of Algorithm 3, if i is selected then the loop is quit and the remaining operations are performed. This means all the remaining squares, the associated multiply and the possible final square operation in case u has a number of bits lesser than W.

So, there will only be some modular multiplication operations out of the main loop. There isn't a function dedicated to an optimized square. With a key composed by 16384 bits the window is set to $W = 5$ and every MPI has a number of limbs equal to $n_{limbs} = 256$.

A good idea is to expose the multiply operations. If it is known when a multiply operations takes place and how much it takes to perform a modular multiplication, then it could also be possible to compute the entire sequence. In the target implementation, every mulitply operation is anticipated by a loop which fetch the factor u from the array of the precomputed values. This loop is always iterated 2^{W-1} times and it performs a conditional copy on a MPI number by invoking the function *_gcry_mpi_set_cond*. This function iterates over the limbs of an MPI independently if the copy effectively takes place or not. This means that before every multiply operation, the body

of the loop contained in $\_gcry\_mpi\_set\_cond$ is repeated $2^{W-1} \cdot n_{limbs} = 16 \cdot 256 = 4096$ times. it is not possible that every iteration will cause a cache miss, but it is reasonable to assume that a good number of cache misses will take place. This portion of code is named c_m. Another idea is to conduce a sort of Performance Degradation Attack on the instructions of the modular multiplication. A cache line associated to a modular multiplication is evicted to conduct the experiment associated in this test case in a more relaxed way. This evicted code fragment is called c_{pda}.

The two portion of code lead the study to consider two types of traces:

1. A base trace collected while only c_{pda} eviciton is applied. This trace is called *base* trace.

2. A trace collected while evicting both $c_{pda}$and c_m. This trace is called *evictm.*

After the portions of code that must be evicted are selected, now the code must be profiled during the execution of the eviction to compute the estimate of the modular multiplication foreach case. To compute these values, *perf* is used to profile the execution of the sign routine with a known key k_{perf} monitoring the clock cycles in user space. Knowing the input key implies that it is also known the number of modular multiplication operations executed by the sliding-window left-to-right square and multiply algorithm. In the case of this test key such a number is n_{mm}. Two analysis were performed:

Trace Analysis Observe the trace of the executing monitoring cycles to find the hotspots of the routine with input k_{perf}. This is needed to find the percentage of cycles that the sign routine spends in executing modular multiplication mm_{perc}.

Counting Analysis This analysis is useful to compute the execution time of modular multiplication. Using k_{perf} and a population size of N_{perf}, it is possible to compute the required value $t_{mm} = \frac{t_{sign} * mm_{perc}}{n_{mm}}$ foreach sample of the population. t_{sign} is the execution time of the sign routine.

This analysis must be performed in both *base* and *evictm* cases. The results of trace analysis showed that the execution of modular multiplications

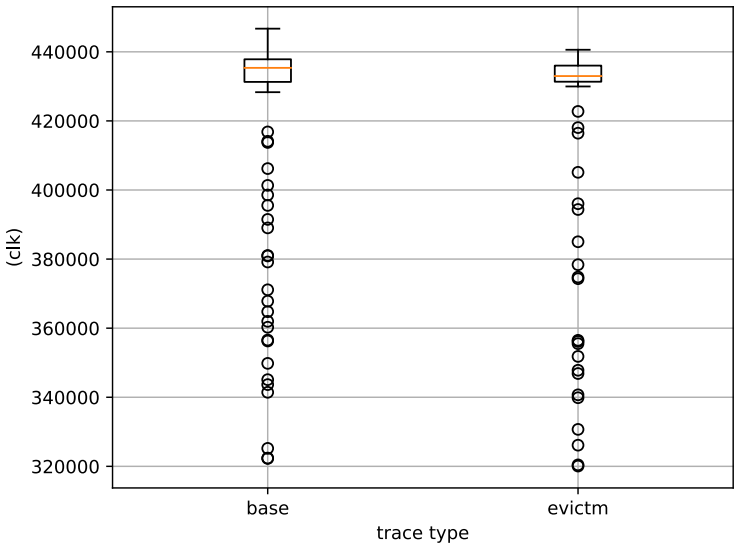

Figure 6.3.3: Boxplot of the counting analysis performed with *base* and *evictm*. There are 25 outlayers in *base* case and 20 in *evictm*.

Table 6.6: Execution time of modular multiplication depending on the type of trace produced by the counting analysis.

eviction type	Mean [*clk*]	Std. Dev [*clk*]
base	428091	24032
evictm	426734	23383

occupates 99.95% of the time in the *base* case and 98.50% in *evictm* case. The mean execution times of modular multiplications of the two used type of traces is shown in Table 6.6 and they were obtained with the counting analysis with a number of $N_{perf} = 200$ samples and $n_{mm} = 19124$. Boxplots of these two populations are shown in Figure 6.3.3. A *t-test* between the two populations produced a *p-value* $p_{mm} = 0.56$. Considering a confidence interval of $\alpha_{mm} = 0.001$ (like in all this study), it is not possible to affirm that the two produced means are different.

Bits	Operations
1	M
0000	SSSS
1	SM
00000	SSSSS
1	SM
000000	SSSSSS
1	SM
...	...

Figure 6.3.4: Example of a forged key with increasing square sequences considering $W = 5$.

6.3.2.2 Comparative Analysis

This is the second final step of the methoology proposed in Section 4.2. It is the step called Comparative Analysis. The forged keys in this step are generated in the way that to every bit set to 0 is associated a square operation s and to every bit set to 1 is associated a square and then a multiply operation sm. This can be easily done be ensuring that between two bits set to 1 there are at least $W - 1$ bits (4 considering this case study) set to 0. Every key has a common initial phase composed by an initial bit set to 1 and then there's a long sequence of 1023 bits set to 0. This produces an initial sequence composed by 10^{1023}. After this sequence there is a bit set to 1, so the result is that there is a multiply operation and then 1024 squares. The reason is that the energy impact of the start-up phase of the process could be considered wrongly as a multiply operation. The first multiply is put at the beginning to avoid the elision of the square operations. After this initial sequence there are 1 bits separated by an increasing sequence of bits set to 0. See Figure 6.3.4 for an example. The pattern is repeated until there aren't enough bits to replicate it, so the remaining bits are set to 1.

Recalling from the quantitative verification, the energy consumption induced by a cache miss is $\tilde{e}_{miss} \cong 0.26\mu J$. Produced traces are obtained by averaging the samples with the same timestamp. Every case has a population of $N_{cs} = 200$ for the sake of having significant results. The traces produced with k_1 are shown in Figure 6.3.5 . Since a modular mutliplication takes

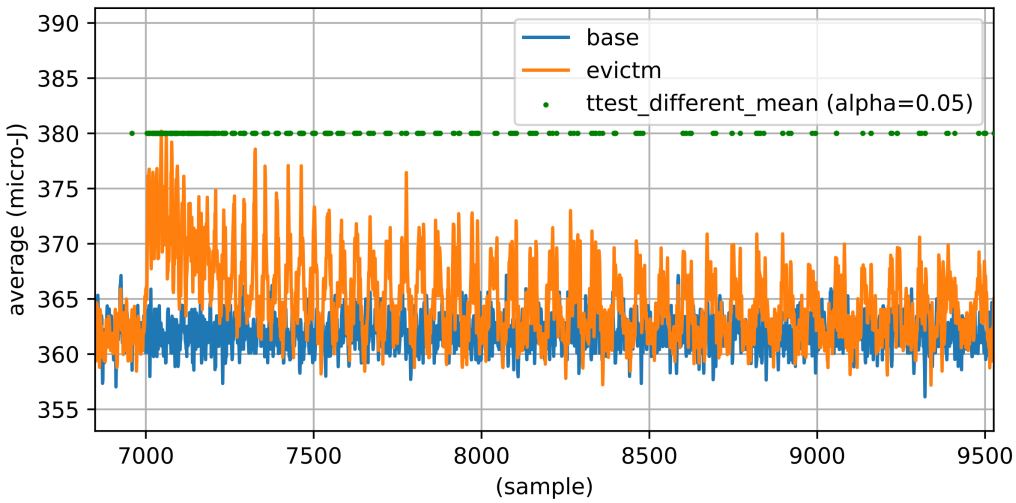

Figure 6.3.5: Traces produced with k_1 and $N_{cs} = 200$ related to DRAM power zone.

about $0.5375ms = \frac{430000clk}{800000\frac{clk}{ms}}$. The difference between the peaks and the *base* energy trace is about $10\mu J$, which means that the system of eviction induces about $\frac{10\mu J}{e_{miss}} = 38.4$ cache misses foreach multiply operation. The next step is identify the minimum sequence of zeroes between two ones that allows to observe 2 distinct multiplies. In theory, the update period of RAPL counters is $0.976\mu s$, so it is possible to observe two multiplies separated by 2 square operations, but this is not possible in the real implementation because:

- Considering a single trace is not tractable. It does not offer any significant information as anticipated in Section 5.2. See Figure 6.3.6 for an example of raw trace of k_1. It is possible to observe some area with a greater energy consumption, but it is not possible to perform any analysis. Vertical resolution iof Intel RAPL counters is well exposed.

- The mean trace contains all the uncertainity offered by the different execution time of the different traces. Two peaks associated to two distinct multiply operations may be merged in a single block of high energy consumption.

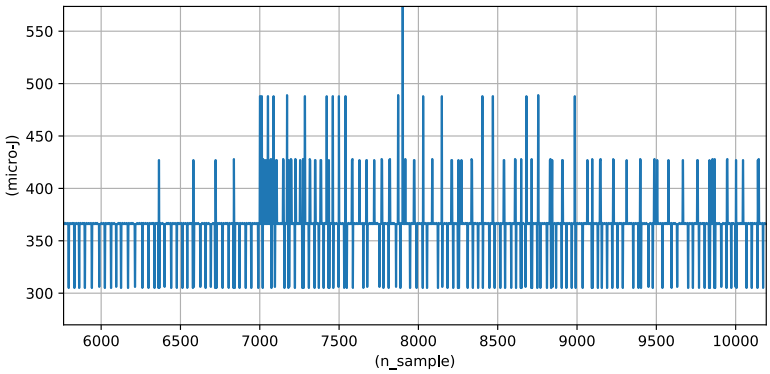

Figure 6.3.6: A raw trace of k_1 of DRAM power zone. It is hard to understand the behaviour of the target program.

To find the minimum sequence of zeroes that allows to distinguish two different multiply, the set of forged key $K_2$is built in the following way:

$$K_2 = \{k \in 10^{1023}(10^n)^k 1^j | n \in \{7, 15, 31, 63\} \wedge 1024 + (n+1) * k + j = 16384\}$$

The keys contained in this set are anticipated by an initial multiply operations and then 1024 squares (just like k_1), then the pattern 10^n is repeated k times (Every key uses a different value of n) until there are not enough remaining bits to repeat the pattern. The remaining bits are filled with ones. The keys forged in this way allows to observe $8, 16, 32$ and 64 square operations between every multiply. The produced traces are the ones in Figure 6.3.7. The last trace is the one with $n = 64$ and it is the one that allows to observe the peaks separeated one by another in the best way. Also the other traces allow to observe peaks in presence of multiply operations, but after a multiplication the energy consumption does not return to the levels of the *base* trace, indicating that in the trace there are effects related to imprecision of the measurement system as discussed before.

In conclusion, in this case study it is possible to observe multiply operations of a RSA sign routine with a private key of 16384 bits. The goodness of the results is limited by the resolution of the Intel RAPL counters. With the used set of eviction and using by collecting a 200 traces it was possi-

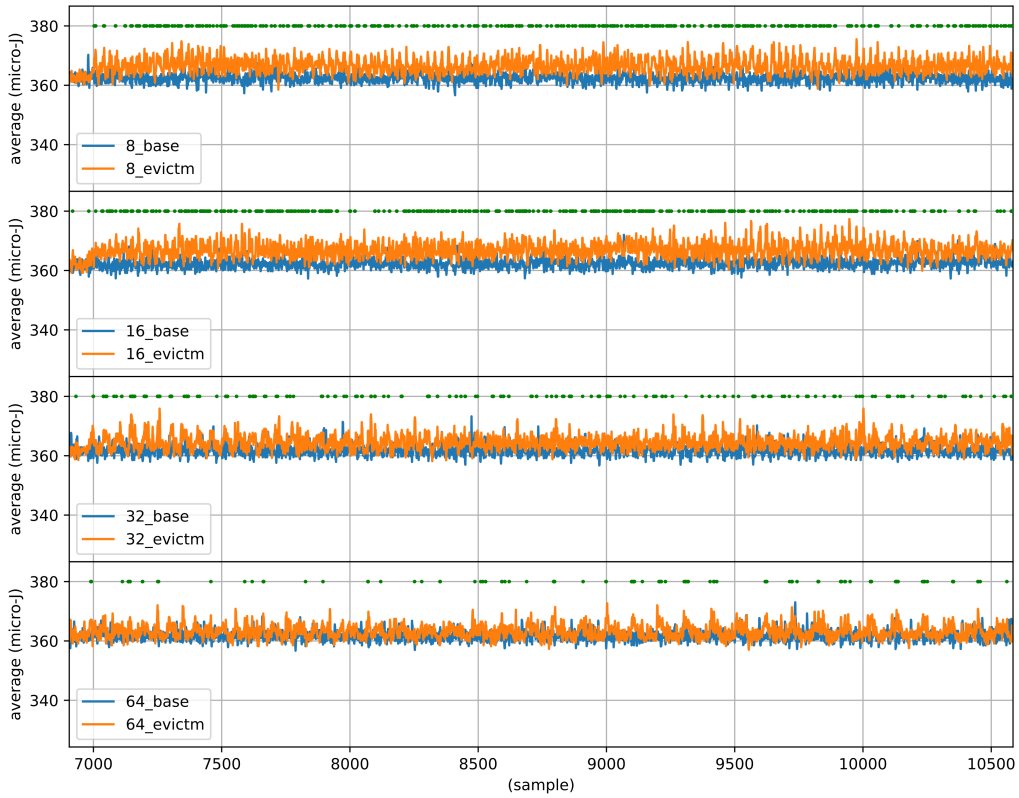

Figure 6.3.7: Results of the keys of the set K_2. Green points indicates different means between populations related to the same sample with $\alpha = 0.005$. A peak represents a multiply.

ble to observe distinct multiply operation in the case they are separated by 64 square operations. The method principally relys on choosing the correct point on which perform the eviction. This fact requires to investigate all the implementation details of the target algorithm.

Chapter 7

Conclusions and Future Works

This chapter reviews the displayed work and offers final conclusion in Section 7.1 by giving an overiew of the proposed methodology and if the results. Section 7.2 describes some future works that may be worth of some analysis after the verification that the target hardware counter leaks information.

7.1 Conclusions

This document proposed a simple but effective methodology to verify if a target hardware counter leaks information about the execution of a set of defined operations. The methodology is based on a simple identification of the operations that may influence the target counter, the construction of the test-cases and then simple statistic analyis with *t-test*s. It also offered to the reader an implementation of this methodology using Intel RAPL as target counters via the Linux Powercap interface. The results demonstrated that Intel RAPL counters does not leak information about arithmetic and logic operations, but it is possible to observe if branches are taken or not and if a cache hit or miss takes place, both in data and instruction cache.

The case study tried employ the information discovered by the implementation of the proposed methodology to collect information about a sign routine of RSA-16384 implemented with *libgcrypt* by looking at DRAM RAPL counter. This case study forced instruction cache misses of target instruc-

tion in a FR style and observed the produced traces in a SPA fashion. It was not possible to look at raw traces due to the low vertical resolution of RAPL counters and the noise present in the whole system, since the RAPL counters collect global values. This forced a shift of the analysis towards a more statistical method by collecting a good number of traces. It was possible to observe multiply operations, but only the case in which they were separated by 64 or more square operations allowed to observe them in a good manner. The results of the case study tells that a SCA based on Intel RAPL counters is not a good solution, since the problems of resolutions of these counters are very low. Moreover, fastest cryptographic algorithm may not suffer information-leak since the perform their operations at a resolution much lower than RAPL counters.

An attack based on Intel RAPL counters could be a good idea in case Intel changes the implementation of these counters with one that produces, at least, more fine-grained temporal results.

7.2 Future Works

The verification of the information-leakage can be easily implemented, so future works can be focused on the possibility of using them to perform a SCA.

The number of PMCs on a modern-architecture is more than 100 (in case of the target architecture is more than 300), so replicating this analysis using the same methodology, but with different hardware counters could be viable of further analysis. For example, all the Intel architectures which support PMCs has an architecural events that counts the cache misses of the last level of cache. It could be possible to perform a Flush+Reload attack, but instead of counting the time to load the target instruction it could be possible to observe if the attacker process suffered of a cache hit or miss by looking at PMCs. This modification of Flush+Reload attack could overcome to some counter-measures proposed to counter it, but it introduces the requirement of using a performance monitoring tools, which they can be disabled by the system administrator.

The case study could also also be re-adapted in a system where hardware energy counters are more precise and/or less system noise is present. This analysis can also be re-used to perform a classic SPA with an oscilloscope in the classic way instead of using these counters.

Since also other processor' producer introduced hardware counters to monitor energy consumption (AMD has APM[4], ARM has a similar support more similar to PMCs [8]), it could be a good idea to investigate these technologies.

Bibliography

[1] CVE-2017-7526. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2017-7526.

[2] PAPI overview. http://icl.cs.utk.edu/papi/overview/index.html.

[3] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 422–435, New York, NY, USA, 2016. ACM.

[4] AMD. *AMD Family 15h Processor BIOS and Kernel Developer Guide*. 2011.

[5] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Jackpot stealing information from large caches via huge pages. *IACR Cryptology ePrint Archive*, 2014:970, 2014.

[6] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *In OLS*, 2009.

[7] The Linux Kernel Archives. Power capping framework. https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt.

[8] ARM. Voltage, current, and power monitoring. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0448i/CHDCEIEJ.html.

[9] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[10] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 555–576. Springer, 2017.

[11] David Chaum. *Blind Signatures for Untraceable Payments*, pages 199–203. Springer US, Boston, MA, 1983.

[12] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.*, 49(C):1162–1174, December 2016.

[13] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale®processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 221–226, New York, NY, USA, 2005. ACM.

[14] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 455–470, New York, NY, USA, 2016. ACM.

[15] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, 2017. `http://www.agner.org/optimize/microarchitecture.pdf`.

[16] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world

applications of curve25519. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 845–858. ACM, 2017.

[17] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks. Shared libraries in sunos. In *Proceedings of the USENIX Summer Conference*, pages 375–390, 1987.

[18] Daniel Gruss, Clementine Maurice, and Klaus Wagner. Flush+flush: A stealthier last-level cache attack. *CoRR*, abs/1511.04594, 2015.

[19] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

[20] Daniel Hackenberg, Robert SchÃ¶ne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *IPDPS Workshops*, pages 896–904. IEEE Computer Society, 2015.

[21] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, January 2012.

[22] Nadia Heninger and Hovav Shacham. Reconstructing rsa private keys from random key bits. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pages 1–17, 2009.

[23] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[24] Clint Huffman. Memory combining in windows 8 and windows server 2012. http://blogs.technet.com/b/clinth/archive/2012/11/29/memory-combining-in-windows-8-and-windows-server-2012.

[25] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-vm RSA key recovery in a public cloud. *IACR Cryptology ePrint Archive*, 2015:898, 2015.

[26] Intel Corporation. Intel xeon processor e5-1600 and e5-2600 v3 product families, volume 2 of 2, register data sheet. https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-v3-datasheet-vol-2.html.

[27] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-035. November 2016.

[28] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253665-061US. December 2016.

[29] Javamex. Rsa key lengths. https://www.javamex.com/tutorials/cryptography/rsa_key_length.shtml.

[30] M. Tim Jones. Anatomy of linux dynamic libraries. https://www.ibm.com/developerworks/library/l-dynamic-libraries/index.html.

[31] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. *Side channel cryptanalysis of product ciphers*, pages 97–110. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[32] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, London, UK, UK, 1999. Springer-Verlag.

[33] libgcrypt. MPI header. https://github.com/Chronic-Dev/libgcrypt/blob/master/src/mpi.h.

[34] The GNU Multiple Precision Arithmetic Library. GMP integer structure. `https://gmplib.org/manual/Integer-Internals.html#Integer-Internals`.

[35] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, 2016. USENIX Association.

[36] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 605–622. IEEE Computer Society, 2015.

[37] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[38] NASM. The netwide assembler. `http://www.nasm.us/`.

[39] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972.

[40] Dag Arne Osvik, Adi Shamir, and Eran 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*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[41] David Oswald. Spa image. `https://www.slideshare.net/phdays/1300-david-oswald-id-and-ip-theft-with-sidechannel-attacks`.

[42] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.

[43] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, March 2012.

[44] Yan Solihin. *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC, 1st edition, 2015.

[45] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2003.

[46] Marc F. Witteman, Jasper G. J. van Woudenberg, and Federico Menarini. *Defeating RSA Multiply-Always and Message Blinding Countermeasures*, pages 77–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[47] Y. Yarom. Mastik: a micro-architectural side-channel toolkit. `http://cs.adelaide.edu.au/~yval/Mastik/`.

[48] Yuval Yarom and Naomi Benger. Recovering openssl ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[49] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732. USENIX Association, 2014.

[50] Younis A. Younis, Kashif Kifayat, Qi Shi, and Bob Askwith. A new prime and probe cache side-channel attack for cloud computing. In Yulei Wu, Geyong Min, Nektarios Georgalas, Jia Hu, Luigi Atzori, Xiaolong Jin, Stephen A. Jarvis, Lei (Chris) Liu, and Ramón Agüero Calvo, editors, *15th IEEE International Conference on Computer and Information Technology, CIT 2015; 14th IEEE International Conference on Ubiquitous Computing and Communications, IUCC 2015; 13th IEEE*

International Conference on Dependable, Autonomic and Secure Computing, DASC 2015; 13th IEEE International Conference on Pervasive Intelligence and Computing, PICom 2015, Liverpool, United Kingdom, October 26-28, 2015, pages 1718–1724. IEEE, 2015.

[51] Yongbin Zhou and Dengguo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.