UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

TESIS DOCTORAL

**Analysis and design of microarchitectural
side-channel attacks and countermeasures**

AUTOR:

Samira Briongos Herrero
Ingeniero de Telecomunicación

DIRECTORES:

José Manuel Moya Fernández
Doctor Ingeniero de Telecomunicación

Pedro José Malagón Marzo
Doctor Ingeniero de Telecomunicación

EXTERNAL MENTOR:

Thomas Eisenbarth
Electrical and Computer Engineering Dr. Ing.

2019

Ph.D Thesis

Título: ANALYSIS AND DESIGN OF
MICROARCHITECTURAL SIDE-CHANNEL ATTACKS
AND COUNTERMEASURES

Autor: SAMIRA BRIONGOS HERRERO

Tutor: JOSÉ MANUEL MOYA FERNÁNDEZ
PEDRO JOSÉ MALAGÓN MARZO

**External
Mentor:** THOMAS EISENBARTH

Departamento: DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

Miembros del tribunal:

Presidente:
Secretario:
Vocal:
Vocal:
Vocal:

Suplente:
Suplente:

Los miembros del tribunal arriba nombrados acuerdan otorgar la calificación de:

Madrid, de de 2019

A mi familia
A mis amigos
A los que están ahí

Acknowledgement

This thesis is the result of several years of effort and multiple cups of coffee. During this time, I have met and worked with wonderful people, all of them have had an influence on me in one way or another, and deserve my gratitude.

First of all, my two advisors are in a great way responsible of this work. Both Pedro Malagón and José Manuel Moya, have given me confidence, support and freedom to achieve the results presented in this thesis. Pedro and his sincerity, have always been there since the very beginning of this work. José and his optimism, has showed me that you can always be there for the people that needs you while being somewhere else trying to make the world a better place.

During these years I have also had the chance of working with Thomas Eisenbarth. I would like to express my gratitude to him for his guidance and for the interesting discussions we have held. I wish him the best of the lucks.

The people in the GreenLSI group have been closely following (and suffering) the development of this thesis. They have always been a great support. Also, the people I have met in the Vernam lab and in Lübeck during my research stays, always made me feel comfortable and treated me as one member of their respective teams.

I also would like to express my gratitude to my football colleagues. They always have led me to keep trying no matter why until you reach your goals, you might fail but effort is not negotiable. Besides, a beer with good friends always help to open your mind.

My whole family and closest friends have been always supporting me during these years. Especially my parents and sisters have always believed in me and cheered me up in the bad moments. They have always taught me that hard work leads to good results. They have had, and will always have a great influence in any success I achieve.

Abstract

Nowadays, computer systems are highly optimized to achieve the maximum performance for the existing fabrication technologies. Each processor includes multiple cores and even multiple threads per core, components that allow it to predict the execution path before it is actually taken, shared resources that are efficiently managed, etc. Indeed, these optimized systems have brought a number of technologies that facilitate our daily tasks such as cloud computing.

However, computer designers have not considered that these microarchitectural improvements can open a back door for malicious agents. Actually, these components that enhance performance introduce, at the same time, measurable side-effects on the system that can be exploited by an adversary to retrieve secret from the processes running in the system. As it has been demonstrated, the hardware can undermine the security of a system. The attacks that exploit the aforementioned side-effects are known as side-channel attacks; in particular, as microarchitectural side-channel attacks.

In the last years, these attacks have gained increasing attention from the research community due to their ability to extract private information from their victims with great resolution in a relatively short time. Besides, these attacks do not damage the hardware in which they run, nor require any special privilege to succeed and are not identified as malicious by most anti-viruses. Researchers have used microarchitectural side-channels to retrieve AES, RSA, ECDSA cryptographic keys among other private information exploiting the shared hardware, even across Virtual Machines (VMs) allocated in the same host.

This research contemplates the analysis of leakage mechanisms in current microprocessors that run different processes in parallel. Specifically, we mainly focus on the Last Level Cache, which is shared across all the cores. Consequently, a malicious process or VM that runs in a core, can use the cache to influence the behavior of any different process or VM running in the same machine to infer information about it.

The main goal of this Ph.D. Thesis is to improve the security of the cyber-physical systems that manipulate and store private information. In order to design proper countermeasures that avoid or minimize the leakage, a complete understanding of the leaking mechanisms and its root causes is required. Thus, in this work, we dive into the details of the cache architecture and successfully

retrieve the replacement policy implemented in Intel processors.

We analyze state of the art cache attacks and demonstrate that they alter the normal behavior of a system. Based on this idea, we develop highly reliable detection-based countermeasures that introduce negligible overhead in the system and that can be utilized by any user at will.

Additionally, this work considers and evaluates other sources of information. For instance, we show how the variations in the performance of a test process can be exploited to determine the total CPU load of the system. We also demonstrate that known attacks have been underestimated by significantly reducing the number of samples required for an attack to succeed. Furthermore, we use the acquired knowledge to design new attacks that circumvent some proposed countermeasures, as detection countermeasures or data prefetching.

Last, this work includes a seminal proposal of a security-aware allocation policy to implement a cloud computing task scheduler. The allocation strategy considers the knowledge acquired during this Ph.D. Thesis, including detection mechanisms, in the consolidation algorithm of Virtual Machines. The policy includes confidence and reputation of the clients and the Virtual Machines to reduce conflicts when collocating them, reducing the risk of suffering a side-channel attack, and acting when an attack is detected.

Resumen

Hoy en día, los sistemas informáticos están altamente optimizados para conseguir el máximo rendimiento de las tecnologías de fabricación existentes. Cada procesador incluye múltiples núcleos e incluso múltiples subprocesos por núcleo, componentes que le permiten predecir la ruta de ejecución antes de que se tome realmente, recursos compartidos que se gestionan eficientemente, etc. De hecho, estos sistemas optimizados han permitido el desarrollo de una serie de tecnologías que facilitan nuestras tareas diarias, como la computación en la nube (*Cloud Computing*).

Sin embargo, los diseñadores de los procesadores no han considerado el hardware una puerta trasera para agentes maliciosos. Como resultado, estos componentes que mejoran el rendimiento introducen, al mismo tiempo, efectos secundarios medibles en el sistema, que pueden ser explotados por un adversario para recuperar información privada de los procesos que se ejecutan en el sistema. Como se ha demostrado, el hardware puede socavar la seguridad de todo el sistema. Los ataques que aprovechan los efectos secundarios antes mencionados, se conocen como ataques de canal auxiliar; en particular, como ataques microarquitectural de canal auxiliar.

En los últimos años, estos ataques han atraído cada vez más la atención de la comunidad investigadora debido a su capacidad para extraer información privada de sus víctimas en un tiempo relativamente corto. Además, estos ataques no dañan el hardware en el que se ejecutan, ni requieren ningún privilegio especial para tener éxito y ni siquiera son identificados como maliciosos por la mayoría de los antivirus. Los investigadores han utilizado canales auxiliares microarquitectural para recuperar claves criptográficas de algoritmos como AES, RSA, ECDSA entre otra información privada explotando el hardware comparado, incluso a través de máquinas virtuales (VMs) asignadas al mismo host.

Esta investigación contempla el análisis de los mecanismos de fuga de información en microprocesadores actuales que ejecutan diferentes procesos en paralelo. Específicamente nos centramos en la caché de último nivel, que es compartida por todos los núcleos. En consecuencia, un proceso malicioso o VM que se ejecuta en un núcleo, puede utilizar la caché para influir en el comportamiento de un proceso o VM diferente, para inferir información sobre él.

El objetivo principal de esta tesis doctoral es mejorar la seguridad de los sis-

temas ciberfísicos que manipulan y almacenan información privada. Para diseñar las contramedidas adecuadas que eviten o minimicen las fugas, se requiere un conocimiento completo de los mecanismos de fuga de información y sus causas fundamentales. En este trabajo demostramos que estos ataques cambian el comportamiento normal de un sistema. Basándonos en esta idea, desarrollamos contramedidas altamente fiables, que tienen como objetivo la detección de los ataques y que introducen una sobrecarga insignificante en el sistema y, además, pueden ser utilizadas por cualquier usuario a voluntad.

Adicionalmente, este trabajo considera y evalúa otras fuentes de información como los efectos en el desempeño de un proceso para determinar la CPU del sistema. Exploramos los ataques conocidos y demostramos que han sido subestimados al reducir significativamente el número de muestras requeridas para completar un ataque. Nos adentramos en los detalles de la arquitectura de la caché y recuperamos con éxito la política de reemplazo implementada en los procesadores Intel. Además, utilizamos el conocimiento adquirido para diseñar nuevos ataques que eluden algunas contramedidas propuestas, como contramedidas que intentan detectarlos o que precargan los datos en memoria.

Por último, este trabajo incluye una propuesta seminal de una política de asignación consciente de la seguridad para implementar un planificador de tareas de un entorno Cloud. La estrategia de asignación considera los conocimientos adquiridos durante esta tesis doctoral, incluidos los mecanismos de detección, en el algoritmo de consolidación de máquinas virtuales. La política incluye los conceptos de confianza y reputación, tanto en clientes como en las máquinas virtuales, para reducir los conflictos al asignar máquinas virtuales a servidores físicos. De este modo, se reduce el riesgo de sufrir un ataque de canal lateral y se actúa cuando se detecta un ataque.

Contents

Abstract	V
Resumen	VII
Glossary	XXIII
1 Introduction	1
1.1 Objectives	4
1.2 Contributions	5
1.3 Publications	6
1.4 Structure of the document	7
2 Background	9
2.1 Virtual Memory	12
2.2 Cache memories	15
2.2.1 Cache organization	16
2.2.1.1 Direct mapped caches	17
2.2.1.2 Fully associative caches	18
2.2.1.3 Set associative caches	19
2.2.2 Inclusiveness property	20
2.2.2.1 Inclusive caches	21
2.2.2.2 Non-Inclusive caches	21
2.2.2.3 Exclusive caches	21
2.2.3 Replacement policies	21
2.3 DRAM	23
2.4 Enhancing performance via prediction	24
2.5 Extended instruction set	25

2.5.1	clflush	26
2.5.2	AES-NI	26
2.5.3	Transactional memory	27
2.5.4	Trusted Execution Environments (TEE)	28
2.5.5	Locked atomic operations	28
2.6	Hardware performance counters	29
2.7	Encryption algorithms	30
2.7.1	AES	31
2.7.2	RSA	32
3	Related Work	35
3.1	Side channel attacks	35
3.2	Microarchitectural attacks	40
3.2.1	Cache attacks	41
3.2.1.1	EVICT+TIME	42
3.2.1.2	FLUSH+RELOAD	42
3.2.1.3	PRIME+PROBE	45
3.2.2	Transient execution attacks	48
3.2.3	Rowhammer attacks	50
3.2.4	Memory Bus Locking Attacks	51
3.2.5	Other microarchitectural attacks	51
3.3	Countermeasures	52
3.3.1	Hardware based countermeasures	53
3.3.2	Detection based countermeasures	53
3.3.3	Preemptive countermeasures	54
4	Reverse engineer of cache Replacement Policies	57
4.1	Intel cache Replacement Policies	57
4.1.1	Advanced Replacement policies	58
4.1.2	Related work	59
4.1.3	Retrieval of Intel cache eviction policies	60
4.1.3.1	Design of the experiments	62
4.1.3.2	Results	64

4.2	Intel Low Level cache Replacement Policies	68
5	Countermeasures	71
5.1	Detection of cache attacks using execution time	72
5.1.1	Case of Study: AES	73
5.1.1.1	Modeling AES	73
5.1.2	Noise tolerance	77
5.1.3	Proposed Detection Algorithm	81
5.2	Detection of cache attacks using hardware performance counters .	84
5.2.1	Performance counters	87
5.2.2	Detection, mitigation and other countermeasures	88
5.2.3	Victim-based Attack Detection	89
5.2.3.1	Analyzing Hardware Performance Events	90
5.2.3.2	Concurrent Signal Assessment	94
5.2.4	CacheShield	95
5.2.5	Evaluation of CacheShield	103
5.2.6	Cache Attack Countermeasures	109
5.3	Detection of Other Microarchitectural Attacks	111
5.3.1	Memory DoS attacks	111
5.3.2	Preliminary study	112
5.3.3	Detection algorithm	114
5.3.4	Evaluation	115
6	Attacks	117
6.1	Inferring the CPU load of a host from a process running on it . .	117
6.1.1	Previous approaches	119
6.1.2	Our approach	121
6.1.2.1	Least absolute shrinkage and selection operator (LASSO)	123
6.1.2.2	Genetic Algorithms	123
6.1.3	Experimental Setup	124
6.1.3.1	Scenarios	125
6.1.3.2	Data acquisition	125

6.1.4	Methodology	126
6.1.4.1	LASSO	126
6.1.4.2	LASSO tuned with GA	127
6.1.5	Results	128
6.2	Improving traditional attacks against AES	132
6.2.1	Attack on AES	133
6.2.1.1	AES fundamentals	133
6.2.1.2	AES T-table implementation	134
6.2.1.3	Attacking AES	135
6.2.1.4	Attack on AES-128	137
6.2.1.5	Attack on AES-256	144
6.2.2	Results	147
6.3	RELOAD+REFRESH	153
6.3.1	Covert channel	157
6.3.2	Noise tolerance	158
6.3.3	Results	159
6.3.3.1	Attacking AES	159
6.3.3.2	Attacking RSA	161
6.3.4	Detection evaluation	164
6.3.5	Discussion of the results	165
6.4	Attack against AES S-Box implementations	166
6.4.1	AES S-Box implementation in OpenSSL	166
6.4.2	Scenario	168
6.4.3	Encryption detection	169
6.4.4	Hitting the last round	172
6.4.4.1	Prime+Abort+Wait+Flush+Reload	172
6.4.4.2	Prime+Abort+Wait+Access+Access	173
6.4.5	Results	176

7 Conclusions 179

7.1	Future work	181
7.1.1	Allocation Policies and Active Monitoring	182

7.1.1.1	Considered threats	183
7.1.1.2	Trust and Reputation Systems	183

Bibliography	185
---------------------	------------

List of Figures

1.1	Global percentage of processors found in Amazon and Azure in 2019. Source: Liftr Cloud Insights	4
2.1	Typical microarchitecture in modern multicore processors.	10
2.2	Block diagram of an Intel Skylake core [1]	11
2.3	Linear-Address Translation Using IA-32e Paging [2]	14
2.4	Example of the memory hierarchy that can be found in current computing systems.	16
2.5	Example of a direct mapped cache.	18
2.6	Example of an associative mapped cache.	19
2.7	Example of a set associative cache (4-way).	20
2.8	Diagram of Linux Performance Tools by Brendan Gregg [3].	29
2.9	Diagram of the AES encryption process	32
3.1	short text	36
3.2	Example of a side-channel scenario. A computer is performing an encryption operation while the adversary tries to obtain the secret key through unintended side channels	37
3.3	Flush+Reload procedure	43
3.4	Complex addressing functions extracted from [4] that determine the slice of the LLC based on the physical address, depending on the number of cores of the processor	46
3.5	Prime+Probe procedure including the three steps required to retrieve the information	47
3.6	Logos of the two vulnerabilities: spectre and meltdown	49
4.1	Distribution of the access times to different data. These times depend on which memory it was located.	64

4.2	Location of the sets controlling the eviction policy within a slice of 2048 sets. Mode 1 (blue) and mode 2 (red).	66
4.3	Detailed representation of the sets with fixed policy within each of the slices for the i7-4790 machine.	66
4.4	Diagram that represents the process of data (D) retrieval whenever the processor makes a request. The blocks with green background represent a cache hit, whereas the blocks with red background represent a cache miss.	67
4.5	Sequence of data access in a cache set updating their content and their associated ages for the two observed policies. Mode 1 of the 4th and 5th generations behaves exactly the same as the 6th 7th and 8th generations. The red arrow points the eviction candidate, that is, the data that would be evicted in case of cache miss. . . .	68
4.6	Tree structure that controls the Pseudo-LRU replacement policy of L1 and L2 caches. The eviction candidate is highlighted	69
5.1	Diagram of the existing layers between the applications and the hardware. The OS provides isolation between processes in the non-virtualized environment whereas the hypervisor provides logical isolation between VMs.	72
5.2	Histogram of times required to read (Reload) a cache line when performing an attack against AES including 40 million of samples. . . .	74
5.3	Distribution of times required for encryptions under different circumstances (attacks flushing different number of lines per encryption or normal operation) for the experiments performed in the Intel Core i7-4790. We use 1 million samples per distribution. . .	75
5.4	Time distribution of encryption in our cloud environment under different circumstances (attacks flushing different number of lines per encryption or normal operation) for the experiments performed in the Intel Xeon E3-1226. We use 1 million samples per distribution. . .	76
5.5	Encryption time histograms with different CPU utilization levels, for the process running normally in the i7 processor.	77
5.6	Distribution of times around peaks for different CPU utilization levels. The process was running normally in the i7 processor. . . .	78
5.7	Encryption time distributions with different memory utilization levels. It includes 1 million samples of the normal execution of the encryption per distribution in the i7 processor	79

5.8	Encryption time distributions of the normal encryption with co-resident virtual machines running the YCSB benchmark in the Xeon E3 machine.	80
5.9	Distribution of times around peaks representing the normal encryption process with co-resident virtual machines running the YCSB benchmark in the Xeon E3 machine.	80
5.10	Distribution of times for encryptions in virtualized environments 100% CPU usage.	81
5.11	Histogram of the number of cycles measured for the decryption process of RSA. From the OpenSSL sliding window implementation and a key length of 2048 bits. It includes 1 million samples and different keys	90
5.12	Mean number of LLC misses over time for AES and RSA executions in the presence and absence of cache attacks. The numbers next to <i>flush</i> indicate the number of lines flushed at a time. After the start up peaks, the misses go to zero in the absence of a cache attack, while under attack they remain high.	94
5.13	Overview of CacheShield.	96
5.14	Relevant parameters for the detection task, the values correspond to an execution of PRIME+PROBE attack on AES	103
5.15	Output of CacheShield when the cache is profiled accessing each set. "1" indicates a positive attack detection.	106
5.16	LLC misses for a complete RSA execution under randmem benchmark. Although there is no attack going on, the randmem benchmark causes multiple misses. In this case, CacheShield does not distinguish whether they are due to an attack or not.	108
5.17	Sample of a normal execution of AES under randmem benchmark. The process is monitored during the time an attack would have taken. Note the differences between the AES and the RSA processes.	108
5.18	CPU utilization of the i7 machine in native environment for different sampling rates in microseconds. Highlighted the 100 us rate as it is the one we use in our experiments	109
5.19	CPU utilization of the Xeon machine in virtualized environment for different sampling rates in microseconds. Highlighted the 100 us rate as it is the one we use in our experiments	109

5.20	Distributions of the values of different counters gathered each 100 ms for different request rates related to the Apache web server application with and without DoS attack executed in the i7 machine virtualized with KVM.	113
5.21	Distribution of the ratios of the values of different counters gathered each 100 ms related to the Apache web server application, and cassandra application with and without DoS attack executed in the Xeon machine virtualized with vmware	114
6.1	Example of the expected CPU load of a system when a process is running or not. The upper figure shows with 0 the instants when it is not running and with 1 the instants when it is running	118
6.2	Histograms of the AES encryption cycles (time) under different CPU loads. Each histogram includes 10000 samples	121
6.3	Outputs of the <i>LASSO</i> regression (10-fold cross validation)	127
6.4	CPU real and estimated load for the Xeon E3 machine	130
6.5	Timing information obtained when attacking AES in the Reload and Probe steps in the test machine. The threshold is shown in red. For the Flush+Reload attack cache misses are represented by times over the threshold. In contrast for the Prime+Probe attack non-access are below the threshold	141
6.6	Key byte possible values with their associated discarding scores. The value marked in red represents the correct key value and is a well distinguishable minimum in the plot	144
6.7	Relation between encryption, straight decryption and Equivalent decryption.	146
6.8	Color map of discarding scores distribution for the 32 bytes of the key retrieved with the Flush+Reload technique. Correct key values are the darkest ones.	150
6.9	Key bits that remain unknown after each encryption for the Flush+Reload technique	151
6.10	Sequence of possible cache set states during the attack for the mode 1 or the newest generations. Note that all elements in the set start with age 2 and end with the same age.	154
6.11	Distribution of the number of misses induced in the victim process by the different attacks, and with no attack. Each includes 1 million samples	161

6.12	Distribution of the encryption times in different situations. Each distribution includes 1 million samples.	161
6.13	Example of a retrieved trace referred to an execution of a RSA decryption. The blue bars represent the real execution of squares (points equal to 800) and multiplies (700). The yellow line represents the information retrieved, low reload times mean detection of the multiply execution.	162
6.14	Detail of a trace of misses measured each 100 μ s for each of the approaches.	163
6.15	Histograms of the Probe times and the encryption times using the S-Box.	167
6.16	Probability of not accessing a line during the last round of the encryption depending on the number of instructions of the last round that have been actually executed.	168
6.17	Diagram of the considered scenario for the attack against the S-Box implementation of AES. The attacker and the victim share the hardware	169
6.18	Percentage of correctly detected encryptions and valid encryptions for an attack as a function of the waiting time measured as the limit of a count. That is, we use a counting thread as clock	170
6.19	General diagram of the process required to evict the data from the cache with just one access.	175
6.20	Key candidates for each of the bytes of the key of the S-Box AES implementation of OpenSSL retrieved using the Prime+Abort + Count + Flush+Reload approach.	177
6.21	Key candidate search space for the 128 bits of the key of the S-Box AES implementation of OpenSSL retrieved using the Prime+Abort+ Count+ Flush+Reload approach.	178

List of Tables

4.1	Details of the machines used in this work to retrieve their Replacement Policies	62
5.1	Probability distribution of cache misses (%)	76
5.2	False positives rates(%) including 1000 request per experiment. . .	83
5.3	False negative rates(%) computed with 1000 request per experiment.	83
5.4	Overview of most relevant hardware performance counters in the presence and absence of attacks, over 1 million calls to RSA and AES, as well as their rankings according to the InfoGain and relief metrics. Level 3 cache misses, PAPI_L3_TCM , clearly have the strongest information for cache attacks.	92
5.5	Mean detection time (ms) per attack and scenario for the evaluated crypto algorithms. Note that in all cases CacheShield has the same configuration and that detection times are much lower than the ones required for the attack to succeed. The mean was computed with at least 1000 samples per scenario	106
5.6	False positive rate for different scenarios and algorithms. (Instances: Y - Yahoo Cloud Serving, V - Video Streaming; R - Randmem)	107
5.7	Mean detection time (s) per attack and scenario for the evaluated cloud applications.	115
5.8	False positive rate for different scenarios and applications. (Instances: Y - Yahoo Cloud Serving, V - Video Streaming; R - Randmem)	116
6.1	MSE, SE and number of features in non-virtualized and virtualized environments for the Xeon E3	128
6.2	MSE, SE and number of features in non-virtualized and virtualized environments for the i7 processor	129
6.3	MSE, SE and number of features in non-virtualized and virtualized environments for the Xeon E5 machine	129

6.4	Experimental platform details.	138
6.5	Mean number of samples required to retrieve the whole key in non virtualized environments	148
6.6	Mean number of samples required to retrieve the whole key in virtualized environments (cross-VM)	148
6.7	Mean number of samples required to retrieve the whole 128 bits key with the access approach (first column), the non-access approach (second column) and the improvement the latter approach means. (V) means virtualized scenario.	149
6.8	Mean times required for gathering the data of Figure 6.9	151
6.9	F-Score for the different attacks when the <i>sender</i> accesses the data at different and fixed intervals (ns)	159
6.10	Mean number of samples required to retrieve each four byte group of the whole AES key when monitoring one line per encryption and their F-Score.	160
6.11	Percentage of samples correctly retrieved and false positives gen- erated by each approach when attacking RSA.	163
6.12	Mean and variance of the different counters collected during the execution of the attacks against AES and RSA. The results were obtained using 10000 samples collected each 100 μs for each scenario.	164
6.13	Experimental platform details.	176

Glossary

ADD	Average Detection Delay.
AES	Advanced Encryption Standard.
AES-NI	Advanced Encryption Standard New Instructions.
ALU	Arithmetic Logic Unit.
ASF	Advanced Synchronization Facility.
ASLR	Address Space Layout Randomization.
BIP	Bimodal Insertion Policy.
BPU	Branch Prediction Unit.
BRRIP	Bimodal RRIP.
BTB	Branch Target Buffer.
CAS	Column Access Strobe.
CFS	Completely Fair Scheduler.
CISC	Complex Instruction Set Computer.
CPA	Correlation Power Analysis.
CPD	Change Point Detection.
CPU	Central Processing Unit.
CSP	Cloud Service Provider.
CUSUM	Cumulative Sum.
DDR3	Double Data Rate type three.
DES	Data Encryption Standard.
DIMM	Dual Inline Memory Module.
DIP	Dynamic Insertion Policy.
DoS	Denial of Service.
DPA	Differential Power Analysis.
DRAM	Dynamic Random Access Memory.
DRRIP	Dynamic RRIP.
DVFS	Dynamic Voltage and Frequency Scaling.
ECC	Error Correcting Code.
ECC	Elliptical curve cryptography.

EM	Expectation Maximization.
EPC	Enclave Page Cache.
FAR	False Alarm Rate.
FIFO	First In First Out.
GA	Genetic Algorithms.
GCM	Galois/Counter Mode.
GPU	Graphical Processing Unit.
HLE	Hardware Lock Elision.
HMAC	Hash-based Message Authentication Code.
HMM	Hidden Markov models.
HPC	Hardware Performance Counter.
IoT	Internet of Things.
ISA	Instruction Set Architecture.
KSM	Kernel Same-page Merging.
KVM	Kernel-based Virtual Machine.
LASSO	Least Absolute Shrinkage and Selection Operator.
LFU	Least Frequently Used.
LIP	LRU Insertion Policy.
LLC	Last Level Cache.
LLR	Log-Likelihood Ratio.
LRU	Least Recently Used.
MLP	Memory Level Parallelism.
MMU	Memory Management Unit.
MRU	Most Recently Used.
MSE	Mean Squared Error.
NIST	National Institute of Standards and Technology.
NRU	Not Recently Used.
OS	Operating System.
PAPI	Performance Application Programming Interface.

PID	Process Identifier.
PIPT	Physically Indexed Physically Tagged.
PMC	Performance Monitoring Counters.
PMU	Performance Monitoring Unit.
PT	Page Table.
RAM	Random Access Memory.
RAS	Row Access Strobe.
RISC	Reduced Instruction Set Computer.
RNN	Recurrent Neural Network.
ROB	Reorder Buffer.
RRIP	Re-Reference Interval Prediction.
RSA	Rivest, Shamir, and Adelman.
RSB	Return Stack Buffer.
RTM	Restricted Transactional Memory.
SBAR	Sampling Based Adaptive Replacement.
SGX	Software Guard Extensions.
SHA	Secure Hash Algorithm.
SIMD	Single Instruction, Multiple Data.
SLA	Service Level Conditions.
SMM	System Management Mode.
SOM	Self Organizing Maps.
SPA	Simple Power Analysis.
SRRIP	Static RRIP.
TA	Template Attacks.
TEE	Trusted Execution Environment.
TLB	Translation Lookaside Buffer.
TME	Transactional Memory Extension.
TPS	Transparent Page Sharing.
TRS	Trust and Reputation System.
TSX	Transactional Synchronization Extensions.
VIPT	Virtually Indexed Physically Tagged.
VM	Virtual Machine.
VMM	Virtual Machine Manager.

Chapter 1

Introduction

Sigmund Freud identified two different drives in human actions. These two drives (impulses or motivations that push a human being to carry out different actions) are contrary but complementary. On the one hand, Eros or the life drive, is the trend towards preserving and creating life. It is related to emotions of love, pro-social behavior or cooperation. On the other hand, Thanatos or the death drive, is the unconscious desire of death and self-destruction. It is related with negative emotions such as fear or anger.

We can observe these complementary behaviors in different ambit of our lives. For example, in the field of information security, some people try to protect the data, to ensure its integrity and confidentiality by designing new ciphers, algorithms or protocols. On the contrary, some other people seek to break these protections. Thanks to this duality, the techniques of both attack and protection have been improved, leading to great advances in the information security field. One cannot exist without the other. This is a similar dichotomy between the reasons behind the development of new technologies: creation or destruction.

In the last years, we have witnessed an abrupt change in society driven by the technology. As one technology emerges, new innovations based on that technology are introduced, and it is soon improved or replaced by a better one. Among others, cloud computing has emerged providing computing resources as a utility, and giving its users the illusion of an infinite pool of resources; smartphones have acquired almost the same importance as desktop computers; the Internet of Things (IoT) is everywhere, allowing the development of applications such as smart homes, smart cities, smart grids, eHealth services, connected vehicles, etc.

All these improvements and changes have been possible due to the rapid increase in transistor densities over the past years, the design of multi-core architectures that outperform single core architectures and allow the execution of

1. Introduction

multiple processes in parallel and an efficient management of computing resources. Indeed, resource sharing is the basis of cloud computing: multiple Virtual Machines (VMs) that can run in parallel and share the same infrastructure, sharing the cost of the service.

As a result of more and more devices interconnected and more services provided, there is an exponential growth in the amount of data generated. This data has to be processed, stored and secured to ensure only authorized users can use it. That is, all these services offer a great benefit, but come with an associated cost in form of new privacy issues and security risks that have to be addressed.

Traditionally, data has been encrypted to protect it from eavesdroppers. Cryptographic algorithms transform the plaintext into the ciphertext using a secret key. When designing these algorithms, it is usually assumed that only the input and the output may be available to an attacker. Many existing encryption algorithms have no known weaknesses, from a logical-theoretical perspective. However, they do not take into account the influence of the environment on them, which can reveal secret information.

Side-channel attacks were first introduced by Kocher [5]. These attacks take advantage of the influence that a computation has on its environment to retrieve information about the data that is being processed. Kocher exploited timing differences observed during the execution of an encryption algorithm to retrieve its secret key. His seminal work triggered multiple research efforts in the area of side-channels. Heat and electromagnetic emissions, power consumption, acoustic emanations and many others are viable sources of information for an attacker.

In particular, computer designers have sought for improved performance with every feature they have added to the Central Processing Unit (CPU). As a consequence, modern computer systems are highly complex and highly optimized. However, designers have disregarded side-channels, and the adversarial effects that each of this features may have on the security of the system. As we will see in this work, the hardware can undermine the security of the software.

Spectre [6] and Meltdown [7] are two recently discovered CPU vulnerabilities that have caused great security upheaval, including emergency patching and general instability. These attacks exploit features such as out of order execution or branch prediction, introduced to improve the performance. They triggered multiple works and, to this day, the research community is not completely sure about the root causes of the leakage and how to avoid it. However, they are only a small subset of a larger category of threats: **microarchitectural side-channel attacks**. These attacks exploit microarchitectural elements, such as cache memories or the DRAM, to recover private information about a victim process. That is, they exploit the microarchitecture with a different objective than the intended one.

Cache memories included in modern computer systems have gained special attention due to their ability to recover fine-grained information about the system. Assuming that an attacker is able to run his own code on the system, he can exploit timing differences observing the victim behavior. Or he can use the timing of his own memory access, which are, in turn, affected by the cache and the code that the victim is running. The attacker can force the cache into a particular and known state; then, he observes how the target application displaces his data; finally, he uses this information to deduce which data the victim is processing. Cache conflicts do not depend on the Operating System (OS) or the hypervisor in cloud environments. They depend on the actual cache design and can bypass the logical isolation provided by the OS or the hypervisor [8, 9].

Cache attacks have targeted cryptographic algorithms since they were discovered [5, 10, 11, 12]. In more recent works, researchers demonstrate leakages using instances of three generic cache attack techniques. Two of them (*Evict+Reload* and *Prime+Probe*) were introduced by Osvik et al [13], and the last one, (*Flush+Reload*), by Yarom et al. [14]. All of them can recover private keys from a victim, so they are considered to be a serious threat. As a result, many well known cryptographic libraries, such as OpenSSL, have changed their implementation of encryption algorithms such as Advanced Encryption Standard (AES), Rivest, Shamir, and Adelman (RSA) or Elliptical curve cryptography (ECC)

Nowadays, information security is taken more seriously than ever. The reputation of an enterprise can be seriously damaged if it is affected by an attack. As a consequence, it is likely that the affected enterprise loses some clients and money. For example, Intel processors have been seriously affected by microarchitectural attacks. According to Liftr Cloud Insights (“a global cloud industry research and advisory service”), these new exploits might accelerate AMD and ARM adoption by cloud providers. And they support this statement on Twitter with the data we include in Figure 1.1. We are not sure if it is the only reason behind Intel losing market share or if there are other reasons behind this phenomenon, such as cost, performance or energy efficiency.

To sum up, in order to secure the increasing amount of data generated, specially in resource sharing scenarios, and preserve user's privacy, it is mandatory to study the threats they are exposed to. That is, to study how the microarchitecture leaks information in order to find proper countermeasures that help to either eliminate those risks or minimize their effects. Note that this study should also investigate whether more powerful and applicable covert channels exist rather than the known ones.

Figure 1.1: Global percentage of processors found in Amazon and Azure in 2019. Source: Liftr Cloud Insights

1.1 Objectives

In this Ph.D. Thesis we mainly focus in one of the aforementioned shared resources: the cache. It is not possible to design and implement useful and effective countermeasures without a detailed knowledge of the microarchitecture, the leaking mechanisms and the threats to which a system is exposed. In this work, we investigate the capabilities of side-channels in order to develop the appropriate countermeasures that prevent their misuse.

Consequently, this thesis has two main objectives:

- Analysis of existing side-channel attacks and evaluation of different sources of leakage in order to understand which are its root causes. To this end, we perform a series of experiments intended to study the effects that one process has on the remaining processes of the system. This knowledge is used to determine, on runtime, if the detected effects correspond to the standard conflicts due to sharing a resource or are due to an intended attack that is compromising the security of a process. That is, we aim to shed light on the understanding of the cache architecture and the leakage mechanisms.
- Design of effective and practical countermeasures. Particularly, we study defensive approaches that a concerned user can adopt without any support from the OS, the hardware manufacturer or the hypervisor in cloud environments. Since it is not possible to change the hardware to completely avoid the leakage, one of our objectives is to minimize this leakage reducing the performance degradation due to the proposed countermeasure.

As a more general objective, this work pursues to improve the overall security

of the system and, as a consequence of that improvement, of cloud computing environments. Moreover, another purpose of this work is to help other researchers to understand the threats to which the systems are exposed, and to extend the existing knowledge of the underlying architecture of computing systems.

1.2 Contributions

During the development of this work, we have performed multiple experiments in realistic environments that have enabled us to identify anomalous behaviors on the system induced by the attacks, to design countermeasures using the available resources, and finally to find different covert channels that existing countermeasures should consider.

In summary, the main contributions of this work are:

- We model cache-attacks against AES using the timers available in the processor. We later use the obtained models to construct a detection system.
- We extend the analysis of the attacks to cover more algorithms and demonstrate that, in absence of an attack, they behave similarly, whereas such behavior changes if they are under attack. We then use anomaly-detection techniques to develop a lightweight tool that is able to accurately detect cache attacks with low overhead.
- We demonstrate that it is possible to drastically reduce the amount of samples required to retrieve an AES secret key using information from cache misses. Besides, we show how our approach can be extended to retrieve AES-256 keys in a practical way with less overhead than the theoretically estimated in the state-of-the-art.
- We evaluate the extent to which the performance of a process is degraded by the existence of other processes and the CPU load they generate. We use the degradation to estimate the CPU load of the physical host. Such information can be used to determine co-residency, or as a covert-channel to transmit information.
- We reverse engineer the replacement policy implemented in modern Intel processors. To the best of our knowledge we are the first ones who provide comprehensive information about the replacement policy and demonstrate that such policy is deterministic. As a result, it is possible to determine and predict which elements are going to be evicted.
- We present RELOAD+REFRESH, a new attack that exploits the replacement policy to gain information from the victim generating a negligible

1. Introduction

amount of cache misses. Since most detection systems consider cache misses as the main source of information, our attack can bypass such systems.

- We show that proposed countermeasures, such as prefetching the data into the cache prior to its utilization, can be circumvented by combining different attack techniques and a deep knowledge of the cache replacement policies. Further, we develop a mechanism that allows an attacker to remove the data from the cache at a concrete instant. We demonstrate the feasibility of our approach by retrieving the secret key of a S-Box implementation of AES

1.3 Publications

The results of this PhD. Thesis, together with other related research, have been published in international conferences and journals. The list of publications can be divided in three major categories. First, publications on new or improved attacks. Second, publications on countermeasures or attack detection techniques and mitigation. Third, publications on machine learning algorithms used to detect attacks based on network packet analysis. This last group of contributions is not included as a part of this thesis.

Attacks:

Samira Briongos, Pedro Malagón, José L. Risco-Martín, José M. Moya. *Building accurate models to determine the current CPU utilization of a host within a virtual machine allocated on it.* SummerSim 2017: 33:1-33:12 (Core B)

Samira Briongos, Pedro Malagón, José M. Moya, Thomas Eisenbarth. *RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks.* (Under review for Usenix Security Symposium 2020 after minor revision, Core A*)

Samira Briongos, Pedro Malagón, Juan-M de Goyeneche, Jose-M Moya. *Cache Misses and the Recovery of the Full AES 256 Key* Applied Sciences 9 (5), 944, 2019 (JCR Q2)

Countermeasures:

Samira Briongos, Pedro Malagón, José L. Risco-Martín, José M. Moya. *Modeling side-channel cache attacks on AES.* SummerSim 2016: 37 (Core B)

Samira Briongos, Gorka Irazoqui, Pedro Malagón, Thomas Eisenbarth. *CacheShield: Detecting Cache Attacks through Self-Observation.* CODASPY 2018: 224-235 (20.9% acceptance rate)

Network attack detection based on Machine Learning:

Roberto Blanco, Juan J. Cilla, **Samira Briongos**, Pedro Malagón, José M. Moya. *Applying Cost-Sensitive Classifiers with Reinforcement Learning to IDS*. IDEAL (1) 2018: 531-538 (Core C)

Roberto Blanco, Pedro Malagón, **Samira Briongos**, José M. Moya. *Anomaly Detection using Gaussian Mixture Probability Model to implement Intrusion Detection Systems*. International Conference on Hybrid Artificial Intelligence Systems (HAIS 2019): 648-659 (Core C)

1.4 Structure of the document

In order to help the reader to understand the results obtained with this work, and to ease the reading of the manuscript, this work is organized as follows:

- *Chapter 2*: provides the background required to understand the attacks that are studied in this work.
- *Chapter 3*: presents existing microarchitectural attacks. We review related works and explain the causes and functioning of state of the art attacks.
- *Chapter 4*: explains the process that we have followed to reverse engineer the replacement policy of Intel cache memories. It first includes detailed information about the last level cache and ends with some observations about the lower level caches.
- *Chapter 5*: summarizes our contributions to the development of counter-measures. We first use time measurements to model and characterize a process under different circumstances and exploit the timing variations introduced by an attacker to detect it. Next, we use hardware performance counters to study how a process should behave and to design an anomaly-based detection system.
- *Chapter 6*: describes our contributions to improve or design new attacks. We start by showing how to retrieve information about the host from a VM running on it. We continue by explaining how to improve the speed of existing cache attacks. Finally, we present two different attacks that use the replacement policy with different purposes. On the one hand RELOAD+REFRESH intends to be stealthy. On the other hand our second proposal (Cache-Sniper) attempts to interrupt a victim process at the desired moment.

1. Introduction

- *Chapter 7:* draws some conclusions and describes some future lines including a proposal for a cloud allocation system that monitors the available resources to get information about the virtual machines and decide the optimum placement considering both energy efficiency and security.

Chapter 2

Background

This work studies and evaluates multiple sources of information leakage in modern processors, with a particular focus on microarchitectural elements. The microarchitecture refers to the detailed internal architecture of a processor and can be defined as the physical implementation of an Instruction Set Architecture (ISA). The ISA defines instruction formats, opcodes, registers, memory addressing and the expected result of the execution of an instruction among other things. These instructions are visible to the programmer. The basic idea of the microarchitecture is to lay out a circuit (including the control unit, the Arithmetic Logic Unit (ALU), the registers, the interconnections between components, buses and others) that can execute the commands defined in an ISA.

Multiple microarchitectures are possible for just one ISA. For example, the x86-64 architecture is used by most modern laptop and desktop computers. Both Intel and AMD processors implement this architecture across various processor generations. However, all of them differ in the underlying microarchitecture. In other words, each microarchitecture is specific for a processor family and model.

This chapter describes how processors are organized and the most relevant microarchitectural concepts required to achieve a better understanding of both the attacks that target them and the countermeasures that can be implemented.

Considered from a high level of abstraction (architectural level) a computer includes a CPU, that processes data and performs control tasks, a main memory, which stores data, and a set of inputs/outputs, which allow it to interact with the outside world. All these components are commonly connected by a system bus. Although each of these main units usually include some common structural components, the concrete microarchitectural elements and its organization depend on the manufacturer and the details are usually hidden for the end-users.

2. Background

Figure 2.1: Typical microarchitecture in modern multicore processors.

A new computer design typically pursues the goal of improving efficiency and offering the best computing performance. The computer designers have included, in every processor generation, new microarchitectural units such as cache memory or Branch Prediction Unit (BPU), and other modules that deal with pipelining, multithreading or multiprocessing in order to achieve these goals. Multiprocessing is implemented with multiple processing units or cores in a single chip, or with various chips in the same motherboard, each of them with multiple cores. Figure 2.1 shows a generic representation of a multicore processor that contains four cores and a shared L3 cache. Figure 2.2 represents a more detailed diagram of the forming blocks of an Intel Skylake core.

A modern computer typically runs multiple processes or VMs concurrently. Although they are isolated theoretically and logically, they are sharing the hardware resources, as they are running on the same piece of hardware. Even if there is a dedicated processing unit, the buses and the memory are shared. In order for this parallelism to be effective and truly improve performance, shared resources must be correctly managed and fairly distributed among all the processes.

Considering a process that is running on a processor, it is using microarchitectural resources. These shared resources might behave differently depending on the data the program is processing, the amount of concurrent processes using them or even on the state of the system. If the use of such resources produces an observable effect, some information on the data being processed might be leaked by the analysis of the behavior of the shared resource. This situation creates an unintended leaky channel known as side-channel. As a consequence, a malicious process can exploit the information leaked to gain unauthorized knowledge on sensitive information that is not obtainable through the main channel.

Figure 2.2: Block diagram of an Intel Skylake core [1]

2. Background

There are multiple side-channel attacks observing information leaked by different shared resources. Although each of them would require an exhaustive analysis, we mainly focus on cache memories and memory hierarchy. In the following sections, we describe them and some other components related to side-channel attacks. For further information about computer architecture and organization, interested readers are referred to [15, 16]. In section 2.7 some of the cryptographic algorithms and their implementations that have been victims of side-channel attacks are introduced.

2.1 Virtual Memory

The memory unit is essential in any computer system, because it stores both the data and the programs that are going to be executed. Ideally, programmers could have access to an unlimited amount of memory that is as fast as the processing units. In real computers, fast memory is expensive so it is commonly organized hierarchically, as a trade-off among the three key characteristics of memory: capacity, access time, and cost. Figure 2.4 shows an example of a hierarchically organized memory. The closest to the processor a memory resides, the smallest, fastest and most expensive per byte it is. In the following sections the cache memories and the main memory are described, preceded by an introduction to the concept of virtual memory.

At the very beginning of the computing history, computers only included a Random Access Memory (RAM) and the secondary memory. Whenever a process was executed, it was loaded into the RAM and programmers had to take care of the management of the memory. As a result of multiple programs running at the same time, the programmer had to split the program into pieces that access to mutually exclusive memory regions and were loaded and unloaded from memory on demand. In that situation there were portions of the memory that were unused and wasted, and the programmer had to deal with the issues derived from the lack of available space in main memory.

Relieving the programmer from that burden was one of the reasons why *Virtual Memory* was designed. With *Virtual Memory* and the utilization of virtual addresses (which were later translated to physical addresses), the OS could automatically manage the two levels of the memory hierarchy and dynamically allocate portions of the program. Later on, the concept of paging was developed. The memory is divided into small sections called pages, and each process is similarly divided. When a process has to be loaded in memory, the takes care of loading the required pages and maintains a Page Table (PT) for each process.

Each process has its own, virtual, address space. The process virtual memory is large and contiguous. However, since the process memory is divided into pages,

each page can be loaded in different places of the physical memory, or even in different memory technologies. Therefore, the *virtual addresses* have to be translated to physical addresses by a combination of hardware and software. This means that the same program can run in multiple regions of the physical memory and that its address space can be longer than the available physical memory. Processes do not longer have access to the physical memory. Indeed, the Memory Management Unit (MMU) maps *virtual addresses* to physical addresses.

In order to translate a *virtual address* to a physical address, the address is split into a virtual page number and a page offset. The size of the page offset depends on the page size. That is, if the page size is p bytes, the page offset corresponds to the lower $\log_2(p)$ bits of the virtual address. The size of the page number in bits is also influenced by the page size: the number of bits of the virtual address minus the number of bits of the page offset. Sizes from 4KB to 16KB are typical. The bits of the page offset remain unchanged and the bits of the page number act as index to the page table from where the physical page number is retrieved.

Current existing paging techniques do not use 64 bit addresses. Instead, they use 48 bits, and such address is called the linear address. Prior to the translation, the processor confirms that the address uses only the 48 bits. Considering page sizes of 4KB (12 bit page offsets), a PT holding all the possible translations for the page number would have 2^{48-12} entries. Instead of a single table, it is common to implement multi-level translation. For example, Intel processors implement page tables with 4 levels, although there is an extension to use 5 level paging already planned [2]. Such extension proposes to extend the addresses from 48 bits to 57 bits. Figure 2.3 shows the process followed to obtain a physical address from a linear address of 48 bits.

Another advantage of *virtual memory* is that it provides logical isolation between processes, becoming the primary mechanism that protects processes from each other. Virtual memory prevents applications from accessing pages that have not been explicitly allocated by them. The most common way of adding such protection to the processes is through the PT. Each table entry includes protection restrictions that determine whether a user can read a page, write or execute that page.

Once a process tries to access a page which is not available yet in main memory, a *page fault* is triggered. The desired page has to be retrieved from a slower secondary storage memory, lower in the memory hierarchy. Additionally, it is possible that another page has to be replaced with this new one. Even if the page is located in memory, each memory access would require two steps. First, the processor obtains the physical address. Second, the processor obtains the data. In order to avoid the latency of the translation, the MMU includes a Translation Lookaside Buffer (TLB) that keeps track of the most recently used translations.

2. Background

Figure 2.3: Linear-Address Translation Using IA-32e Paging [2]

The design of the TLB is based on the idea that once a translation is used, it will be probably used again in the near future. So the TLB acts as a cache that hold these translations and it is usually located between the cache and the RAM main memory [17] although it can also reside between the CPU and the cache or between the different levels of the cache. We also distinguish between TLB hits when the requested address is in the TLB and TLB misses on the opposite situation. After a TLB miss the PT has to be checked (*page walk*).

Shared memory

As we have already stated, programs have to be loaded from hard disk drives to main memory to be executed, using RAM for both instructions and data, as the access time to RAM is much faster than to hard disk. RAM memory is a limited resource, so when there is no free memory available, some data and instructions are moved to what is referred as a swap area: a copy of RAM stored in the hard disk. When this area has to be used in an application, its performance is severely affected and degraded.

Operating systems employ mechanisms such as memory sharing to reduce memory utilization. For instance, considering that there are two different processes using the same library, instead of loading it twice into RAM memory, sharing memory capabilities allow to map the same physical memory page into

the address spaces of each of the processes.

Deduplication is a concrete method of shared memory, which was originally introduced to improve the memory utilization of the Virtual Machine Manager (VMM) in virtualized environments, and was later applied to non-virtualized environments. The hypervisor or the operating system scans the physical memory and recognizes processes that place the same data in memory; that is, pages with identical content. When two or more pages happen to include the same content, all the mappings to these identical pages are redirected to one of them, and the other pages are released. However, if any change is performed by any process in the merged pages, memory is duplicated again.

The Linux memory deduplication feature implementation is called Kernel Same-page Merging (KSM) and appeared for the first time in Linux kernel version 2.6.32. KSM is used as a page sharing technique by the Kernel-based Virtual Machine (KVM). KSM scans the user memory for potential pages to be shared. Since it only scans potential candidates, it avoids monitoring the whole memory continuously [18].

The deduplication optimization saves memory allowing more virtual machines to run on the host machine. To exemplify this statement, we refer to the kernel-newbies community[19], that states that it is possible to run over 50 Windows XP VMs with 1GB of RAM each on a machine with just 16 GB of RAM. In terms of performance, deduplication is an attractive feature for cloud providers. However, after several demonstrations of side-channel attacks exploiting page sharing, they are advised to disable this feature and, to the best of our knowledge, no cloud provider is ignoring this advice.

2.2 Cache memories

Figure 2.4 depicts the improved memory hierarchy introduced to close the speed gap between processing units and memory access. Cache memories are one of its main components. They were included in the hierarchy because processors were working at higher frequencies each time, whereas the DRAM used for the main memory was not keeping up. As a consequence, it was becoming a bottleneck. Since the DRAM is too slow compared to the processor and fast memory is expensive, the economical solution adopted by designers was to include the cache in the memory hierarchy. This hierarchy takes advantage of the principles of both temporal and spatial locality. That is to say, most programs do not access data or code uniformly, and data that has been recently accessed is likely to be accessed again soon. Similarly, data residing in memory regions close to the data that is being used is also likely to be accessed.

2. Background

Figure 2.4: Example of the memory hierarchy that can be found in current computing systems.

Thus, CPU caches are small and fast memories located between the CPU and main memory, specially designed to hide main memory access latencies. They contain a copy of portions of the main memory that has been recently used. When the processor requests some data, it first checks whether the data is located in the cache hierarchy or not. If it is (*cache hit*), then it is retrieved from the cache reducing the latency observed by the processor due to the memory access time. On the contrary (*cache miss*), this data is loaded from main memory and stored in the cache replacing data already stored on it if necessary. The average access time is determined by the cache access time, miss rates and miss penalties. Consequently, the overall performance of the system is a result of the cache design.

Modern processors usually include multiple levels of cache memories; low level caches (L1 and L2) are core private, smaller and closer to the processor, whereas the last level cache (LLC or L3) is bigger and commonly shared among all the cores. The size of the caches and its number, the mapping function that determines how lines from main memory are placed in the cache, the cache organization, and the replacement policies are key design decisions with great impact on the performance of the system. In the following, we describe in detail these basic concepts.

2.2.1 Cache organization

In section 2.1 we have already discussed how virtual addresses are translated to physical addresses and how the memory that a process uses is related to the physical memory. In this section we describe how memory blocks are placed in the cache based on their address. Considering a main memory of size 2^n , each element can be accessed with a unique address of n bits. Each of these elements will, in

turn, be placed in the cache when requested by the processor. To exploit spatial locality, instead of loading word by word, a block of fixed size (usually 64 bytes) containing multiple words is placed in the cache when necessary. Consequently, the cache is organized in blocks of fixed size called lines. Since the cache is smaller than the main memory, multiple blocks of the main memory can be placed in the same cache line. There are two questions that have to be solved by the designer at this point: how do we know if certain data is cached or not, and where in the cache is going to be placed the data from main memory.

Regarding to the first question, a *tag* is used to distinguish between memory blocks and to identify whether they are cached or not. Commonly, the *tag* uses the most significant bits of the n -bit address which are not used as part of the cache address. Additionally, each cache line usually includes some control bits that indicate, for example, if it is valid or not, or if it has been modified since loaded.

So as to the second question, designers usually define a mapping function between main memory blocks and the cache, using part of the bits of the block address. This mapping function determines the organization of the cache. Depending on it, we can distinguish three main designs: direct mapped, fully associative or set-associative caches. We briefly describe each of them in the next paragraphs.

We can distinguish between physically or virtually indexed caches, depending on the address used to locate their contents. Note that physically indexed caches have to wait for the translated address, whereas virtually indexed caches use directly the virtual address used by the process. Similarly, there are virtually and physically tagged caches. Low level caches, which are meant to be faster, are usually Virtually Indexed Physically Tagged (VIPT) whereas the last level caches are Physically Indexed Physically Tagged (PIPT). For instance, Intel L1 cache is said to be VIPT, whereas L3 is PIPT.

2.2.1.1 Direct mapped caches

This is the simplest mapping function, it assigns each block of main memory to exactly one line in the cache. That is, this cache design establish a relation one to one between main memory locations and cache lines. The relation between a memory address and its placement within the cache is depicted in Figure 2.5. Assuming that a cache line can hold 2^m blocks or lines, and that each line holds 64 bytes (2^6), the cache holds $2^{(m+6)}$ bytes. The lowest ($o = 6$) bits of the memory address, represent the offset and will be used to locate the data within the block. The following m bits select the cache line (the position) where the data will be placed. Finally, considering that the main memory has size 2^n and

2. Background

Figure 2.5: Example of a direct mapped cache.

it is consequently addressable with n bits, the remaining $n - m - o$ bits will be the *tag* that will be used to check whether the data is cached or not.

Since the processor only has to look in one location to check if the data is in the cache or not, and this location is determined by the address, the search process is very efficient. Direct mapped caches are also cheap and easy to implement. On the other hand, there are multiple memory lines that map to the same cache line (2^{n-m} lines). They are named *congruent lines*. This means that, if the processor has to work repeatedly with two or more different and congruent addresses, this kind of cache will experience misses most of the time, reducing the hit ratio and decreasing performance.

2.2.1.2 Fully associative caches

Contrary to direct mapped caches, fully associative caches allow the processor to load each memory block into any position of the cache. This approach eliminates the main disadvantage of direct mapped caches, as there are no congruent lines. This cache also loads memory data into cache lines or blocks, so the least significant bits (offset) will be similarly used to locate the data within the line. Again, assuming a line size of 64 bytes, and a main memory size of 2^n bytes, the remaining $(n - 6)$ bits of the address form the *tag*. Figure 2.6 shows an example of an associative cache.

Note that all the *tags* of all the elements allocated in the cache have to be checked in parallel whenever the processor requests any data. To do that, this kind of cache memory includes a control logic based on multiple comparators that increases the hardware cost. As a consequence, fully associative placement is only practical for caches with small number of lines.

Figure 2.6: Example of an associative mapped cache.

Once the cache memory is filled with data, the following access to a different piece of data not cached will generate a miss. Since the blocks of memory can be located anywhere, these caches have flexibility to decide which line to replace, but they try to maximize the hit ratio. Section 2.2.3 includes a discussion on the different replacement policies.

2.2.1.3 Set associative caches

Direct mapped caches can suffer many cache misses, whereas associative caches require complex logic. As a consequence, set associative cache memories appeared as a trade-off. They are an intermediate solution between the previous ones. These caches are divided into multiple sets, each of them containing a fixed number of lines. Figure 2.7 shows an example of a set associative cache.

A W -way set associative cache holds W lines in each set (4 in the example). The number of sets depends on the size of the cache and the associativity. If each line holds 64 bytes, and the cache size can hold 2^m of these blocks, the number of sets $2^s = 2^m / (W)$. As in the previous cache designs, the main memory address of the piece of data requested by the processor is used to select the set in which the data is placed. The least significant bits of the address are used to locate the data within the line, the following s bits are used to select the set, and the remaining $(n - s - 6)$ bits are used as the *tag*. With the set number extracted, there is a limited number of locations (W) where each block can be placed, which have to be checked in parallel to find the requested memory address.

A direct mapped cache can be seen as a one-way set associative cache in which the number of sets is the same as the number of lines the cache holds and each element in the main memory maps to the only line in the set. Similarly, a fully

2. Background

Figure 2.7: Example of a set associative cache (4-way).

associative cache is just an 2^m -way set associative cache where 2^m represents the number of lines of the cache.

Most modern processors have adopted this elegant and practical solution and include set associative caches. The number of ways is a design decision; usually, increasing the associativity decreases the miss rate, but it slightly increases the cost and the access time. For example, the Intel Core i5-7600K has two 32KB 8-way L1 caches (one for data and the other for instructions), a 256KB 4-way L2 cache and a 6MB 12-way L3 cache.

2.2.2 Inclusiveness property

As stated before, modern computers usually include multiple levels of cache memories. Further, in multiprocessor architectures, each core can have private caches and share another cache with the remaining cores. In this scenario, multiple copies of the same data can exist in different caches. If processors are allowed to access and modify this data and no extra precautions are taken, they could end up seeing two different values for the same location. This is known as the coherence problem, and it is usually handled in hardware.

If the program modifies a piece of data during its execution, it will be first updated in the L1 cache. At this moment, the values in the cache and in the main memory are inconsistent. In order to address this situation *write-through* caches always write the data into both the memory and the cache. On the other hand, a *write-back* scheme only writes data to memory when the modified block is replaced from the cache. In this scenario, the processor maintains a write buffer

to store the data that will be updated in the memory and the L1 cache deals with fast writes. This approach improves performance in exchange for a more complex implementation.

When designing the cache, there are different approaches referring to the inclusiveness property that can make cache coherence much easier to implement.

2.2.2.1 Inclusive caches

These caches require that any memory block that is located in the low level caches, i.e. L1 and L2 caches, has also to be present in the Last Level Cache (LLC). This property itself simplifies the coherence problem: the LLC, which is shared across all the cores, tracks any update on the data. Although the LLC might not have the exact copy of the data that is being manipulated in the L1 cache (*write-back* caches), it knows which address is cached. The drawback of this approach is that multiple lines of the hierarchy can be wasted holding the same data. Intel processors have traditionally adopted LLC inclusive caches.

2.2.2.2 Non-Inclusive caches

They do not have any particular requirement about where each memory block has to reside. As a consequence, a memory block can be located at multiple cache levels at a time, even in different cores. This approach makes coherence harder to implement. The newest Intel Skylake Server microarchitecture uses a non-inclusive Last Level Cache [20], probably due to cache attacks or seeking an improvement in performance.

2.2.2.3 Exclusive caches

In this design, a memory block can only be loaded at one cache level at a time. Consequently, implementing the coherence protocol is a little easier than in the previous case. However, it has to consider all the caches in the memory hierarchy. The advantage in this case is that both the usable space and efficiency increase. For example, some of the ARM processors include exclusive caches.

2.2.3 Replacement policies

Another key decision in the design of the cache is the replacement policy. We have already stated that, after a *cache miss*, the data that has been loaded from the main memory is placed into the cache. It also means that, in case the cache is full, one of the currently cached lines is evicted and replaced with this new

2. Background

line. A good replacement policy, the algorithm that decides which memory block is evicted to make room for the new one, is crucial for maximizing the hit ratio and achieving good performance. Some examples of well known possible policies include:

- First In First Out (FIFO): it acts as a FIFO queue; it evicts the first element inserted in the cache regardless of its actual utilization.
- Least Recently Used (LRU) or pseudo-LRU: these algorithms remove the least recently used line from the cache; i.e., the data block that has not been used for a longer time is evicted. Thus, these policies require to keep track of the utilization of the blocks and the times in which they were used.
- Least Frequently Used (LFU): it is based on the frequency of the access to the blocks located in the cache. Access to the blocks are counted, and in case of conflict, the element with fewer registered uses is the one evicted.
- CLOCK [21]: it was originally designed for page replacement. It keeps a circular list of the elements. It points at the “oldest” element in the list. In the event of a cache miss, it checks if the value of a parameter R of that element is equal to 0. In this case it evicts that element. On the contrary, it resets the value of R to 0, and the pointer advances one position. The process is repeated until an element with R equal to 0 is found. If one element is accessed, its corresponding R is set to 1.
- Pseudo-random: it randomly selects one candidate to evict when necessary. This means there is no need to track the access, which results in a very simple algorithm that has been implemented in some ARM processors.

LRU replacement policies are hard to implement, since they require assigning a timestamp for each access, or updating an ordered list with each access, and this also means complex hardware. This is the reason why many approximations to LRU are implemented in modern high performance processors. A known pseudo-LRU policy is Not Recently Used (NRU) [22]. NRU only requires one bit per cache line. This bit is set whenever a cache line is accessed. If setting a bit implies that all the bits of a cache set are equal to one, then all the bits (except for the one that has just being accessed) are cleared. When a cache line has to be replaced NRU selects one with the control bit equal to zero.

Another approximation to implement pseudo-LRU policies are binary trees. In this policy, each element of the set is located in each leaf of the tree. In the tree, each node (which splits in two branches) has a 1-bit flag that indicates whether the oldest element is located following the right or the left branch depending on its value. Thus, to find the eviction candidate, the tree must be traversed from

the node origin, according to the values of the flags in the nodes, till the leaf is reached. Whenever an item is accessed, or loaded into the cache after a miss, the values of the nodes that have been traversed to retrieve it, must be switched. This way they end up pointing to the opposite direction.

LRU or pseudo LRU policies have demonstrated to perform well in most situations. Nevertheless, LRU policy behaves poorly for memory-intensive workloads whose working set is bigger than the available cache size, or for scans (bursts of one-time access requests). Different policies, such as LFU, outperform LRU when dealing with scans. If it were possible to choose between these two policies depending on the situation, performance would be enhanced. This is how adaptive algorithms came up. Such algorithms are capable to adapt themselves to changes in the workloads. Usually, they decide which policy to use by observing the performance of the processor for the different policies. This adaptive behavior has been observed in Intel processors [23, 24].

2.3 DRAM

Typically, modern computer systems implement the main memory as Dynamic Random Access Memory (DRAM). Compared with the memory cells of cache memories, these cells need significantly less area within the chip; in exchange, they have higher latencies. Traditionally, RAM memories are volatile; that is, they need to be powered constantly, or the data they hold would be lost. Particularly, DRAM cells store the data as the electrical charge on a capacitor. Based on a comparison with a threshold, it is possible to decide whether the capacitor is loaded or not, which is interpreted as a binary 1 or 0. Capacitors tend to discharge, so refreshing the data they hold is necessary to keep it stored.

Modern processors have a memory controller which is able to communicate through the memory bus with the DRAM. DRAMs are usually organized as a matrix of rows and columns. They use a two-level decoding structure: one half of the address is sent during the Row Access Strobe (RAS) and the remaining half is sent during the following Column Access Strobe (CAS). Memory controllers determine the sequence of commands to send over the memory bus to the RAM and also include some hardware to refresh the DRAM. They are able to refresh all the bits in a row simultaneously. This refreshing capability is due to the DRAM organization in rows. Moreover, in order to boost performance, rows are buffered to speed up repeated access to the same row. All the data of the last accessed row is kept in the buffer until a different row is accessed.

The rows of the can be in two different states: they are either *closed* or *opened*. We call a row hit the situation in which the currently opened row has the data the processor wants to retrieve, so the controller can just fetch it from the buffer.

2. Background

On the contrary, a row conflict occurs. In this situation, the current row has to be closed and the data in the buffer has to be written back to the cells. The next step is to activate the row with the desired data and load it in the row buffer. As expected, row conflicts increase access latencies.

Modern computer systems organize the DRAM into channels, Dual Inline Memory Modules (DIMMs), ranks (set of DRAM chips connected to the same chip select) and banks (groups of rows with their own row buffer). For example Double Data Rate type three (DDR3) DRAM memory has 8 banks per rank and 1 to 4 ranks, since each of these banks is independent, multiple rows can be opened at the same time. The memory controller has to schedule memory access to take advantage of the bank/rank parallelism and to maximize row hits.

The architecture of the DRAM is prone to some errors also known as soft errors. These errors do not destruct the DRAM and happen at random times. For example, if there are power supply problems, or maybe some radiation affects the chip, one of the bits can flip its value. In order to deal with these unwanted changes, some modules include an Error Correcting Code (ECC). It requires extra bits and memory controllers and it usually uses the Hamming code. Therefore, an ECC memory module is able to correct 1-bit errors and to detect 2-bit errors. Note that the mainboard and processor are required to support error correction for ECC; otherwise, the modules still run without providing ECC.

2.4 Enhancing performance via prediction

Processors are currently designed to achieve the greatest possible performance. As decoding and executing an instruction requires several cycles, modern processors improve efficiency by using a processing pipeline to overlap the execution of multiple instructions. Each instruction is divided into a series of steps that can be processed in parallel by different units increasing the throughput of the system. If the next instruction is unknown or data is not ready, there is no input data for the processing pipeline and it blocks. Thus, processors try to exploit instruction parallelism and to reduce the moments when they are stalled waiting for one step to complete.

When running a program, this situation can happen when it reaches a point in which the following instruction, the one that has to be executed next, depends on the output of the current instruction; for example, when reaching conditional, indirect, direct, call, and return branches. For this reason, current architectures include a BPU which enables the processor to “speculatively” execute some subsequent instructions before the outcome of the branch has been obtained. If the guess is correct, the processor will have already computed the following instructions by the time it is asked to. On the contrary, the processor discards the

calculated results and continues its normal execution as if the pipeline had done no operation.

This unit is the result of the combination of different ideas. First of all, the microarchitecture uses a dynamic prediction strategy to decide which instructions to execute when a branch instruction is reached. This strategy is based on the recent execution history, which is tracked with a Branch Target Buffer (BTB). Whenever a branch instruction that is going to be executed is found in the BTB, the prediction unit bases its decision on the history for that entry. If the branch is taken, the history information is updated. If the instruction is not in the BTB, its address is loaded into the BTB replacing an older entry if necessary.

Secondly, it is based on speculation. The processor speculates on the outcome of branches and executes the program as if its guesses were correct, time before control dependencies are resolved. As a result, the processor requires mechanisms to handle the situation where the speculation is incorrect; it has to be allowed to undo the effects of an incorrectly and speculatively executed sequence, and continue its normal operation with the data available prior to the speculative execution.

Note that, in order to reduce stalled cycles, instructions might be executed out of order. However, they are forced to commit in order to prevent any irrevocable action. This fact requires additional hardware (the Reorder Buffer (ROB)) that holds the results of instructions that have finished but not have committed.

2.5 Extended instruction set

From a programmer's point of view, the ISA defines any knowledge required to program a computer. As we have already stated, it provides the interface between the hardware and the lowest level software, defining instruction formats, instruction opcodes, registers, instructions to handle data and memory, the expected effects of executing certain instructions in both registers and memory, and so on and so forth. There are different types of instructions; some of them are commonly included in different instruction sets, such as data handling and memory operations, arithmetic and logic operations, control flow operations or co-processor instructions.

As we have already stated, the microarchitecture defines the set of techniques used to implement the instruction set. Processors with different microarchitectures can share a common instruction set (Intel and AMD implement versions of the x86 instruction set while having completely different designs) which they should be able to interpret. There are multiple ways of classifying an ISA; for example, considering the architectural complexity we define Complex Instruction

2. Background

Set Computer (CISC) or Reduced Instruction Set Computer (RISC).

For some applications or features designed for modern computers, the ISA turns out to be insufficient. Some examples are current operating systems, virtual memory, concurrent I/O and enhanced hardware security features among many other features. In these cases, the ISA is extended to provide support for these additional tasks. Indeed, the x86 instruction set has been extended many times, introducing wider registers, data types and new functionality. Some of the extensions of the instruction set which are relevant in this work are briefly explained in the following sections.

2.5.1 `clflush`

The `clflush` instruction was introduced with the SSE2 extensions. According to the specification, this instruction invalidates the cache line that contains the linear address specified with the memory operand. And such cache line is effectively removed from every level of the cache hierarchy. That is, calling `clflush m` flushes the cache line containing *m*. Only the `mfence` instruction orders this instruction. The `clflush` can be available even without SSE2. It has its own CPUID flag, which is used to check its availability.

This instruction is one of the reasons why side channel cache attacks such as FLUSH+RELOAD exist. Intel processors let any user execute the `clflush` instruction. On the other hand, ARM processors sometimes do not even implement the instruction or, if they do, it is only accessible for privileged users.

2.5.2 AES-NI

The Advanced Encryption Standard (AES) is the standard for symmetric encryption adopted by the U.S. National Institute of Standards and Technology (NIST) in 2001. The standard is detailed in [25]. Data is encrypted in blocks of 128 bits (16 bytes) using different key sizes (128, 192, or 256 bits). Depending on the key size the encryption is performed in 10, 12 or 14 rounds.

One of the operations that is performed at the beginning of each round is `SubBytes`. It mainly uses a table (S-Box) to replace each byte of a 4x4 matrix with a new one. The next operation is `MixColumns`, which multiplies a constant matrix with the resulting array of `SubBytes`. The Software implementations of the AES algorithm use precalculated table lookups to obtain the value of either `SubBytes` (S-Box) or both `SubBytes` and `MixColumns` (T-tables). The drawback of this approach is that the access is key dependant and, consequently, different cache lines are accessed during the encryption or decryption process depending on the key and the data. Cache attacks or timing attacks can easily exploit this

dependency to obtain the secret key.

Processor manufacturers have decided to face and resolve the memory access leakage problem of AES software implementations by adding a set of six Single Instruction, Multiple Data (SIMD) instructions to the x86 instruction set. The Advanced Encryption Standard New Instructions (AES-NI) also improve the speed of the AES encryption and decryption processes. Currently, Intel, AMD and some of the ARM processors include this set of instructions (AESENC, AESENCLAST, AESDEC, AESDECLAST, AESKEYGENASSIST, AESIMC).

2.5.3 Transactional memory

When multiple threads are running in parallel and try to access a shared resource, a synchronization mechanism is required to avoid conflicts. Transactional memory is an attempt to simplify concurrent programming, avoiding common problems with mutual exclusion and improving performance, since the processor is not locked. The programmer is no longer responsible for identifying the locks and the order in which they are acquired; she only needs to identify the regions which are going to be defined as part of the “transaction”. As a result, developing software which uses shared large data structures, uses many cores and scales with its number should be easier.

Transactional memory enables optimistic execution of the transactional code regions specified by the programmer. The processor executes the specified sections assuming that it is going to be possible to complete them without any interference. During the execution of these parts, the variables and the results of the operations are only visible for that thread; that is, any update performed in these regions is not visible to other threads. If the execution ends successfully, the processor commits all the changes as if they have occurred instantaneously and they become visible to the remaining processes.

If, on the other hand, the transaction is unsuccessful, the processor cannot commit the updates atomically. Consequently, the processor requires a mechanism to undo all the updates, discard all the changes, and restore the architectural state to pretend that the execution never happened. Unsuccessful transactions are commonly known as transactional aborts.

Intel Transactional Synchronization Extensions (TSX) are the Intel's implementation of hardware transactional memory. TSX provides two software interfaces: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). AMD, on the other side, named Advanced Synchronization Facility (ASF) to its proposed extension to the x86 instruction set that implements hardware transactional memory. Even ARM has recently announced that they are going to support the Transactional Memory Extension (TME) [26].

2. Background

2.5.4 Trusted Execution Environments (TEE)

Intel Software Guard Extensions (SGX), as well as the ARM TrustZone, are extensions to the architecture designed to deal with the increasing concerns about privacy and confidentiality. SGX assumes that only the hardware can be trusted, whereas applications, OSs or hypervisors can be compromised. Consequently, it provides an environment (the Trusted Execution Environment (TEE)) for applications to run isolated, called the *enclave*. The *enclave* is not only isolated from other applications running on the same system, but also from the Operating System or from the hypervisor. That is, *enclaves* protect sensitive data from unauthorized access or modification even when attempted by software running at higher privilege levels.

Developers can benefit from the hardware isolation provided by the *enclaves* for security critical applications or parts of their code. *Enclaves* can even be delivered as part of an untrusted application. Any untrusted application can only communicate or interact with the enclave using specific and predefined interfaces.

Memory pages that belong to an *enclave* are encrypted in DRAM and protected from possible snoopers. These memory pages can only map to a reserved DRAM region, known as the Enclave Page Cache (EPC), and are only decrypted when processed by the CPU. Whenever an *enclave* function is called by a non-trusted application, only the code within the *enclave* can see its own data, and any request of external access would be always denied. For further details about SGX we suggest the reader the work of Costan et al. [27].

2.5.5 Locked atomic operations

Atomic operations are not an extension of the instruction set per se. They are rather instructions that feature a behavior that makes them somehow special.

Atomic operations (i.e. not splittable in parts) operate in different variables so that any other thread either sees the final value of the variable or its previous value. Traditionally, Intel used to provide a LOCK signal that could be used as a prefix on different instructions which, in turn, forced a global bus lock. This fact was degrading the performance of the system. For this reason, Intel switched to cache-locking, where cache lines are locked instead of the shared bus.

However, Intel still locks the bus when dealing with certain operations known as *exotic atomic locked operations*. These operations include locked access to uncacheable memory, locked access to data that spans two cache lines (misaligned access) and page-walks from an uncacheable page table. It is likely that both AMD and Intel support this behavior for legacy reasons. Regarding ARM, it also supports atomic operations through two operations: LDREX and STREX.

2. Background

It is possible to monitor and count instructions executed, cycles, cache access or misses, branches correctly predicted or branch misses among many other events. As the PMU allows detailed insight into the state of the processor in real-time, it is a valuable tool for debugging applications and their performance. Indeed, they were originally intended to find bottlenecks and to improve programmer's code, and consequently the global performance of a piece of software running on a determined machine.

There are multiple libraries, such as Performance Application Programming Interface (PAPI) or `oprofile` [29, 30], that simplify the task of configuring and reading these registers. Similarly, there are many other utilities that have been developed for Linux, that allow to monitor the performance of the different parts of the system. Brendan Gregg, who has been conducting computer performance analysis for years, has developed a useful diagram (Figure 2.8) including these tools [3].

2.7 Encryption algorithms

The previous sections introduced some microarchitectural components and techniques which are used or mentioned along this work. This section is related with the main encryption algorithms that are similarly used or mentioned in this work. The implementation of encryption algorithms is a typical target of side channel attacks. A side channel attack on an unsecured implementation might be more efficient in retrieving the secret key than brute force algorithms applied to the direct channel.

There are different types of cryptographic algorithms in use, which traditionally have been classified as symmetric cryptography and public-key cryptography.

Symmetric cryptosystems are sometimes also called private key algorithms, and include algorithms for encryption, authentication as well as hashing. Encryption and authentication schemes use single key for both the encryption/authentication and decryption/verification. Popular algorithms include AES and Data Encryption Standard (DES) for encryption, Secure Hash Algorithm (SHA) (SHA-2 and SHA-3) for hashing and Hash-based Message Authentication Code (HMAC) or Galois/Counter Mode (GCM) for authentication or authenticated encryption. Symmetric primitives are usually heavily optimized for performance and feature constant execution flows. However, some implementations make use of table look-ups, which often result in exploitable cache leakage. One example is AES, the most widely used encryption algorithm. For AES, table look-ups are difficult to avoid, unless hardware support such as AES-NI is available.

Public key cryptosystems use a public key for encryption or verification and private key for decryption or signing. While public key cryptography can be used in more flexible ways, the used primitives are much more costly than for symmetric cryptography. As a result, public key cryptography is mainly used for authentication and key exchange to establish a communication session, where payloads are protected using symmetric cryptography. Another important offered service are certificates, which require digital signatures for generation and verification of certificates. RSA, ECC and ElGamal are currently the prevailing schemes for public key cryptography.

The following subsections offer some details on two algorithms, AES and RSA, one of each type, used in the contributions of this work.

2.7.1 AES

As we have already mentioned, the AES algorithm is the standard for symmetric encryption. It is a block cipher which consists of a repeated application of a round transformation on the state, a 16-byte block denoted S . Both encryption and decryption behave similarly. The number of iteration rounds, N_r , depends on the size of the key: 10 rounds for 128-bits, 12 rounds for 192-bits and 14 rounds for 256-bits. The concrete mathematical details of the AES are not relevant for the purpose of this work, but interested readers are referred to [25]. The T-table-based OpenSSL implementation is the preferred victim in successful published attacks [31] and is often used for comparison.

The encryption process is depicted in Figure 2.9. The normal round transformation in the encryption process consists of four steps denoted: **SubBytes**, **ShiftRows**, **MixColumns** and **AddRoundKey**, being **SubBytes** the only non-linear transformation. **SubBytes** is an S-box S_{RD} applied to the elements of the state. **ShiftRows** is an element transposition that rotates the rows of the state i positions to the left. **MixColumns** operates on the state column by column. The operation is a matrix multiplication of a 4x4 known matrix and each of 4 the original columns. **AddRoundKey** is the addition (bitwise XOR in $GF(2)$) of the elements of the state with the corresponding element of the round key.

In each round, the algorithm uses a different round key K^r , which obtained from the algorithm key using a known and fixed scheme. Once a round key is known, there is a straight forward algorithm to recover the same amount of bits of the encryption key. The first and last round of AES are slightly different than the others. They consist of an XOR operation between some values of the lookup T-Tables and the respective round key.

The T-Table implementation of AES uses tables with pre-calculated values

2. Background

Figure 2.9: Diagram of the AES encryption process

to reduce the number of operations performed during each round (**SubBytes**, **ShiftRows**, **MixColumns** and **AddRoundKey**) and to improve performance of the process. As a result, each T-Table is accessed at least once each round. When the encryption ends, the lines of the T-tables used by the encryption process are available in the cache memory. Each of these T-Tables holds 256 values of 4 bytes each, so they occupy various cache lines.

On the other hand, the S-Box implementation does not merge two operations into one. The S-Box is the table that holds the data for the *SubBytes* operation. This implementation only uses one table with 256 byte values (the S-Box) that is continuously accessed during the encryption process.

Nowadays, the default AES implementation in most cryptographic libraries, uses by default (if available) the AES-NI extension mentioned in section 2.5.2..

2.7.2 RSA

RSA is the most widely used public key cryptographic algorithm. Imagine that Alice and Bob want to establish a secure communication. Each of them is assumed to have two keys; a public key (e,d), which is publicly known and a private key (n,d) which remains secret. If Alice wants to send an encrypted message to Bob, she will have to encrypt the message with Bobs public key. Once Bob receives the

Algorithm 1 Square-and-Multiply exponentiation

Input: base b , modulo m , exponent $e = (e_{n-1} \dots e_0)_2$

Output: $b^e \bmod(m)$

```

 $r \leftarrow 1$ 
for  $i$  from  $n-1$  downto  $0$  do
     $r \leftarrow r^2 \bmod(m)$ 
    if  $e_i = 1$  then
         $r \leftarrow r \cdot b \bmod(m)$ 
    end if
end for
return  $r$ ;

```

encrypted message he will be the only one able to decrypt it with his private key. In practice, RSA takes advantage of the practical difficulty of the factorization of large numbers. It uses two prime numbers (p, q) to compute $n = p \cdot q$. The value n will be part of the public key while the prime numbers will be used to generate secret key. Even if the attacker knows the value n , he will not be able to retrieve p and q .

The mathematical details referred to the algorithm are out of the scope of this work. In order to understand the attacks only the encryption and decryption operations are relevant. Being m the message, the ciphertext c is obtained as $c = m^e \bmod n$ and it is recovered with an analogous operation $m = c^d \bmod n$. The modular exponentiation operation that uses the secret key, is the target of the attacks. There are multiple ways of implementing this exponentiation [32, 33]. However we will particularly focus on two: the square-and-multiply and the sliding-window exponentiations.

The square-and-multiply approach is summarized in the algorithm 1. It scans the bits of the secret exponent performing an square operation independently of the value of the scanned bit, and a multiplication if such bit is equal to 1. Thus, an attacker monitoring the square and multiply operations can retrieve the sequence of bits of the exponent.

The pseudocode of the sliding-window exponentiation procedure is represented in the algorithm 2. The exponent is scanned to obtain a sequence of windows w_i of maximum length W . Usually W is a value between 4 and 8. The procedure to obtain the windows is as follows: when scanning the exponent from the most significant to the least significant bit, the windows will be composed by the maximum sequence, up to W bits, starting in a 1 and ending in a 1. If after that sequence there is a 0, then there will be a window with just one zero. The window is moved and the procedure repeats until all the bits of the exponent have been scanned. Once the exponent is scanned, the exponentiation procedure is analogous to the square-and-multiply algorithm. The square operation is per-

2. Background

Algorithm 2 Sliding-window exponentiation

Input: window size W , base b , modulo m , N -bit exponent e as n windows of length $L(w_i)$

Output: $b^e \bmod(m)$

\\Precomputation phase

$g[0] \leftarrow b \bmod(m)$

$s \leftarrow \text{MULT}(g[0], g[0]) \bmod(m)$

for j from 1 to 2^{W-1} **do**

$g[j] \leftarrow \text{MULT}(g[j-1], s) \bmod(m)$

end for

\\Exponentiation

$r \leftarrow 1$

for i from n downto 1 **do**

for j from 1 to $L(w_i)$ **do**

$r \leftarrow \text{MULT}(r, r) \bmod(m)$

end for

if $w_i \neq 0$ **then**

$r \leftarrow \text{MULT}(r, g[(w_i - 1)/2]) \bmod(m)$

end if

end for

return r ;

formed for each window and if the window is a non-zero value, it also performs a multiplication. The main difference is that in order to perform the multiplication, an element stored in a pre-computed table is accessed. Consequently, the access depends on the value of the window, and as a result these key dependent access can be used to leak the key.

Related Work

Confidentiality is one of the principles of information security. Data is encrypted and information protected so it can only be seen or used by people who are authorized to access it. Unauthorized users may try to retrieve this information applying different techniques. Brute force techniques, or phishing, are typical attacks that try to break confidentiality using direct channels. A different approach involves monitoring unintended channels from which the attacker can deduce or obtain the secret information. These channels are known as side-channels. A side-channel attack does not attack an algorithm itself (which can be theoretically secure and most likely will not leak any information via the *direct observation* of its input and output) ; it exploits the information gained from the implementation of the algorithm via indirect side-channels.

This work mainly focuses on microarchitectural side-channel attacks. Chapter 2 describes some of these microarchitectural elements that are susceptible to contribute to implement one of these attacks. This chapter provides first a general overview of different side channels, to show the overall idea of this kind of attacks. Later, it provides a detailed explanation of current microarchitectural attacks and countermeasures.

3.1 Side channel attacks

When talking about side channel attacks, we mean that an attacker is retrieving information from a system by some means or ways not intended for that purpose. For example, figure 3.1 depicts the general idea of a side channel. A safecracker is trying to open a safe using a listening device to get extra information about the state of the lock. He will probably manipulate the lock until he is able to hear

3. Related Work

any different noise that reveals the key.

Figure 3.1: Example of a side-channel on a safe-box¹

There are multiple classes of side channels, and they are usually classified depending on the source of leakage. Typical side channels include power consumption, electromagnetic emanations, optical, acoustical channels, or timing analysis. In 1996, Kocher [5] introduced a new idea: an attacker with accurate timing measurements of a private cryptographic and non-constant time operation may be able to break an implementation of cryptosystems such as RSA or Diffie-Hellman. This work was an inflection point in the field of cryptography research.

As pointed out by Standaert in [34], a cryptographic primitive can not only be viewed as a model or black box that transforms some input into an output depending on a secret key. Since the program will run on a device, on a processor or on a given environment, it may present specific characteristics or physically observable phenomena caused by the aforementioned execution. Side-channel attacks are less general than traditional cryptanalysis; however, they are much more powerful. Consequently, they must be considered by manufacturers and designers. Figure 3.2 shows an example of a side-channel scenario in which the attacker monitors the leakage emanated during an encryption process to retrieve the secret key.

¹This figure was retrieved from <https://www.ibtimes.co.uk/meltdown-melted-down-everything-except-one-thing-1663785>

Figure 3.2: Example of a side-channel scenario. A computer is performing an encryption operation while the adversary tries to obtain the secret key through unintended side channels

Timing attacks

Cryptosystems can take different amount of time to process the data. Timing attacks exploit these timing variations to derive information about the secret key. For example, an attacker can measure how much time requires a computation with a known key, then he can measure this time for the victim's process and finally compare both times. Different instruction execution times, cache hits, misses, or branches taken are some examples of factors that can introduce timing variations.

Kocher [5] showed that both RSA and Diffie-Hellman private operations compute the modular exponentiation $y^x \bmod n$, where x is the secret exponent. A particular implementation of the modular exponentiation operation executes different number of instructions depending on the secret values of the exponent. As a result, the total execution time is a function of the private exponent.

Later, Brumley and Boneh [35] mounted a time attack against a remote OpenSSL-based server targeting a different implementation of RSA that included Chinese Remainder and Montgomery reductions which were believed to be invulnerable to timing attacks. Also targeting OpenSSL, but this time an implementation for elliptic curves over binary fields, Brumley et al. [36] mounted a timing attack that recovers the ECDSA private key with information about a small number of signatures.

Bernstein [12] performed a timing attack against an implementation of AES and was able to successfully retrieve the secret key of a server running on a different computer. His work was based on the observation that memory-access times can leak secret information. Concretely, he exploited variations stemming from cache access. His work showed, on the one hand, that the timings of the lookup access to an array entry stored in memory depend on the entry index. On the other hand, that due to this variations out of the control of the programmer,

3. Related Work

it is really hard to write high-speed constant-time code for general computers.

Optical attacks

The basic idea of these attacks is simple: secrets and sensitive information can be recovered by some visual mean as a high resolution camera or a similar device that has such capabilities. Ferrigno et al. [37] showed that when the logical state of a transistor changes, a few photons are emitted. They employed a device able to detect these photons and recovered an AES key of an implementation running on a PIC16F84A. Particularly they targeted the initial round of AES `AddRoundKey` and were able to distinguish the key bits at the time they were XORED with the plaintext.

Acoustic cryptanalysis

The key observation in this case is that the different elements of the system produce sound during the execution of a piece of software as do other devices such as the keyboard [38]. These attacks exploit the acoustic emanations from a system to recover secret information. Backes et al. [39] recovered up to 95% of the English text printed with a dot-matrix printer. They first identified and extracted the relevant features from the captured traces emitted by the printer. These traces were recorded with a microphone placed in the proximity of the printer. Then, they built a model for different words and, in a second phase, they used that model to recognize new sounds and recover the printed text.

More impresively, Genking et al.[40] were able to extract full 4096-bit RSA keys from the GnuPG software during the decryption of some chosen ciphertexts. To this end, they recorded the sound generated by the computer (this noise is due to vibration in some mechanical components) using a plain mobile phone or a microphone. When they analyzed the traces, they observed that the acoustic signature of modular integer exponentiation depends on the modulus involved. That is, each RSA signature has a unique acoustic signature, so secret keys can be distinguished using acoustic information.

Electromagnetic attacks

Microprocessors radiate electromagnetic fields during the execution of computing tasks. As Standaert et al. [34] mention, the electromagnetic radiation is data dependant in CMOS devices. As a result, an attacker can use measurements of the electromagnetic field to infer cryptographic keys.

The leakage can be theoretically explained by the Biot-Savart law. This law shows the relation between the data and the field, and between the current direction and the field orientation. Based on these observations it is possible to derive a leakage model. For example, the Hamming distance model is a model which assumes that the Hamming distance of switching values is correlated with the side-channel leakage. Peeters et al. [41] provide further information about how to get accurate leakage models for CMOS devices. Agrawal et al. [42] conducted an investigation of EM side-channel leakage from chipcards to study and assess the magnitude of EM exposure.

Power attacks

Electronic devices or microprocessors consume power when performing their assigned tasks. Like the electromagnetic field, power consumption can also be measured and used as a side channel to get secret information. Power consumption is one of the side-channels that has received more attention as it is a practical mean to obtain secret information. Particularly important is the dynamic power consumption, which is due to the charge and discharge of capacitances, since it determines a relationship between the data that the device is processing and the power consumption measured.

The attack process starts with a thorough analysis of the target device and its implementation of the victim algorithm. Then, the following step to implement a practical attack is getting the measurements. A good measurement setup, including an acquisition device (typically an oscilloscope), is crucial and, if carefully prepared, can reduce the amount of information required to get the secret information.

Last but not least, the attacker needs to process the traces. Traditionally, literature classifies the attacks according to the way the traces are treated. For example, Kocher et al. [43] distinguish between Simple Power Analysis (SPA) and Differential Power Analysis (DPA), although many other techniques emerged after his seminal work.

- SPA [44, 43] attempts to interpret the power consumption of a device and deduce information about its performed operations with a few traces, even just one. Sometimes even the visual inspection of the trace is enough to get the desired information.
- DPA is much more powerful than SPA. It does not require detailed knowledge about the attacked device and it is harder to prevent. DPA uses statistical analysis and error correction techniques on multiple side-channel traces. This attack was proposed by Kocher et al. [43] and is based on the

3. Related Work

hypothesis that internal state values depend on the key. The attacker tries different keys and checks which ones matches with the collected traces at the desired instants (when the states depend on the key).

- Correlation Power Analysis (CPA) attack is somehow similar to a DPA attack except for the fact that it uses the correlation factor method instead of the difference of means. This technique was introduced by Brier et al. [45] in 2004. They identified two trends regarding to statistical power analysis that had been used so far: DPA and other proposals that use the correlation factor between the collected power samples and the Hamming weight of the data being manipulated by the device [46, 47]. They examined the limitations and weaknesses of both approaches and came up with an approach based on the Hamming distance model “aiming at either improving the Hamming weight model, or enhancing the DPA itself”.
- Template Attacks (TA) assume that similar devices leak in a similar way. These attacks require a profiling step aimed to model the leakage of the training device. Once this initial step has finished, the obtained model will be used for the actual attack. Chari et al. [48] first proposed this attack and presented a multivariate Gaussian model of the signal and noise as the *template* of the computation.
- Stochastic Methods, as the proposed by Schindler et al. [49], came out as a new approach to optimize the efficiency of differential side channel cryptanalysis. Indeed, using such methods the efficiency of power attacks can be increased up to a factor of fifty. In [49] authors approximate the leakage function using the maximum likelihood principle.

3.2 Microarchitectural attacks

As in the previous cases, microarchitectural attacks exploit different characteristics of a processor's microarchitecture. Again, we refer the reader back to chapter 2 in case some details about the different microarchitectural elements that can leak private information are not clear. In this section, we discuss the state-of-the-art microarchitectural attacks, and in section 3.3, we discuss some countermeasures.

When analyzing and studying microarchitectural attacks, it is quite important to identify which of the components included in the processor can serve as a covert channel: how do they leak information, how much information do they leak and how to avoid such leakage if possible. Depending on the characteristics of the component, we observe that it is possible to construct single-core, cross-core or even cross-processor attacks [50].

3.2.1 Cache attacks

Cache memories, as well as improving performance, create new covert channels that can be exploited to extract sensitive information. When a process tries to access some data, if it is already loaded into the cache (namely a cache hit), the time required to recover this data is significantly lower than the access time in case the data has to be retrieved from main memory (cache miss). Therefore, when a cryptographic process access some data, the execution time will depend on the locality of the accessed data: more precisely, on the presence or not of the data in the cache memory. This time information can, as a consequence, be exploited to gain private information from cryptographic processes, such as secret keys.

As stated before, cache attacks monitor the utilization of the cache to retrieve information about a co-resident victim. Memory access carried out by one victim process can be spied and deduced by an attacker process exploiting the underlying shared hardware. Thus, whenever the pattern of memory access of a security-critical piece of software depends on the actual value of sensitive data, such as a secret key, the consequences of the attack can be as devastating as an impersonation of the victim. Ideally, programmers should write constant-time code that uses all the available resources uniformly.

Traditionally cache attacks have been classified as:

- *Time driven attacks*: the information is learned by observing the timing profile for multiple executions of a target cipher.
- *Trace driven attacks*: the information is gained by monitoring the cache directly. The attacker is assumed to have access to the cache profile when running the target process.
- *Access driven attacks*: the information is retrieved from the sets of the cache accessed during the execution of the target process.

It was in 1992 when the cache memory was first mentioned as a covert channel to extract sensitive information by Hu [51]. Later on, in 1996, Kocher [5] introduced the first theoretical attacks, as did Kelsey [52] in 1998 describing the possibility of performing attacks based on cache hit ratios. Years later, in 2002, Page [10] studied a theoretical example of cache attacks for DES, as would do Tsunoo [11] for the same algorithm investigating timing side-channels created due to table lookups. In 2004, Bernstein [12] proposed the first time-driven attack after observing non-constant times when executing cryptographic algorithms. Although the attack he presented was not practical, his correlation attack has been investigated extensively and even retried between VMs recently [53].

3. Related Work

Soon, another attack was proposed by Percival [54], who suggested that an attacker could determine the cache ways occupied by other processes by measuring the access times to all the ways of a cache set. The reason is that the access times are correlated with the number of occupied ways. In that work, he exploited the L1 data cache, which is shared among the processes running in one core. Indeed, he exploited the hyper-threading technology introduced years ago by Intel. Each core can execute two virtual threads, which share the resources of the core, maximizing the number of computations performed in parallel.

In 2006, Osvik et al.[13] proposed two techniques, which have been widely used since then: EVICT+TIME and PRIME+PROBE. These techniques are intended to allow an attacker to determine the cache sets accessed by a victim process. He exploited the inter-process leakage in the L1 cache to retrieve an AES secret key. A significantly more powerful attack that exploits shared memory and the Completely Fair Scheduler (CFS) was proposed by Gullasch [55] . The same principles of the attack were later exploited by Yarom and Falkner [14], who named the attack FLUSH+RELOAD. The target of this attack was the L3 cache, as it is inclusive and shared between cores. From these attacks, the FLUSH+RELOAD and the PRIME+PROBE (and their variants) out-stand over the rest due to their higher resolution.

Among all the microarchitectural attacks presented in this work, this thesis focuses mainly on cache attacks. For this reason, the following subsections provide a detailed description of the main attacks that target the cache. The FLUSH+RELOAD and PRIME+PROBE sections include the fundamental concepts needed to understand the basis of each attack before reviewing its related works.

3.2.1.1 EVICT+TIME

This technique consists of three steps: first, an encryption is triggered and its execution time measured; second, an attacker evicts some lines from cache; third, the encryption time is measured again. By comparing the second time with the first measure, an attacker can decide whether the cache line was or not accessed, as higher times will be related to the use of the mentioned line.

3.2.1.2 FLUSH+RELOAD

This technique relies on the existence of shared memory, i.e. victim and attacker use the same data during their respective execution, and on the existence of the `clflush` instruction in the target processor. In this scenario, if the attacker *flushes* the desired lines from the cache, he can be sure that, if the victim process needs to retrieve the *flushed* line, it will have to load it from main memory.

Since Intel caches are inclusive, whenever a process needs to use a piece of data that is not in the cache, it places it in the LLC once retrieved from the main memory. As a result, that data is available in the cache for any other process accessing the same memory address, as the LLC is shared among all the processors.

The attack

FLUSH+RELOAD also works in three stages that are depicted in figure 3.3:

1. The desired lines from the cache are flushed by the attacker (Flush). Since the attacker needs to know exactly which lines or addresses can leak information, some reverse engineering prior to this step may be required to retrieve that information.
2. The attacker waits for a predefined amount of time while the victim runs its process or a fragment of it. Depending on the scenario the attacker can trigger the execution of the victim process.
3. The previously flushed line is accessed by the attacker (Reload), measuring the access time. Depending on the reload time, the attacker decides whether the line was, or not, accessed by the victim. Low reload times mean that the victim has used the data while high reload times is the opposite.

Figure 3.3: Flush+Reload procedure

This attack does not require a great knowledge of the underlying architecture; it just uses the `clflush` instruction to remove the target lines from the cache in the initialization stage. It is easy to implement, quite resistant to micro-architectural noise, and provides exact information about the data actually used

3. Related Work

by the victim. For these reasons, it has become really popular. However, its main drawback is that it requires memory deduplication to be enabled. As stated before, deduplication is an optimization technique designed to improve memory utilization by merging duplicate memory pages. Another implication of such requirement is that it can only recover information coming from statically allocated data. Shared memory also assumes that the attacker and the victim have to be using the same cryptographic library.

The FLUSH+RELOAD attack was introduced by Gullasch et al. [55]. In that work they exploited a similar technique to retrieve an AES key. But it was the work of Yarom et al. [14] the one which extended the attack to target the LLC, gave a name to the attack and demonstrated that just one properly collected trace is enough to retrieve an RSA key.

The attack is also applicable in virtual environments, where it is able to break the logical boundaries between virtual machines allocated in the same host. In such environment, Irazoqui et al. [31] showed that it is possible to retrieve an AES key from a T-table based implementation of the algorithm. Gülmezoglu et al. [56] extended their approach to target more realistic scenarios. Further, Zhang et al. [8] showed that the technique is applicable in several commercial PaaS clouds.

In virtual environments, in order to retrieve information referring to neighboring VMs, the attacker had to achieve co-residency with the victim. This was possible, as demonstrated by Ristenpart et al. [9] who used ping timings to decide whether two instances were running on the same host or not. Additionally, he took advantage of the co-allocation to perform a keystroke timing attack.

Benger et al. [57] performed some experiments aiming to extract information from OpenSSL ECDSA signature requests. Their target was the curve secp256k1 used in the Bitcoin protocol. Yarom et al. [58] also used FLUSH+RELOAD against ECDSA. The attack also recovered TLS protocol session messages bringing back to life *Lucky 13* [59]. It found application in the detection and identification of cryptographic libraries [60] used by neighboring VMs. A different approximation is assuming that the attacker does not have prior knowledge of the library version that he is going to attack. Based on that assumption, Gruss et al. presented the Cache template attack [61]. They enforced the existence of shared memory between the attacking and the attacked processes to automatically profile the victim.

When the FLUSH+RELOAD attack was suggested [14], researchers believed it was not applicable to ARM architectures. However, some posterior works demonstrated that these processors are not exempt of this threat. Zhang et al. [62] studied the channel in Android devices that mount ARM processors, and similarly Lipp et al. [63] mounted LLC attacks on ARM devices. A year later,

Green et al. [64] discovered a feature included in some ARM processors that could make cache attacks against ARM harder.

Relying on the `clflush` instruction and with the same requirements (shared memory) as FLUSH+RELOAD, Gruss et al. [65] proposed the FLUSH+FLUSH attack. It was intended to be stealthy and bypass the attack monitoring systems [66, 67] that have been designed by that time. The main difference with FLUSH+RELOAD is that this variant recovers the information by measuring the execution time of the `clflush` instruction instead of the reload time, thus avoiding direct cache access on the attacker side. This was the key fact to avoid detection. Gruss et al. noticed that when the data is loaded into the cache, it takes longer to *flush* it out of the cache. This timing difference is then exploited to retrieve AES keys. As a result, this work triggered more research on countermeasures, and recent works have demonstrated that it is detectable [68, 69].

3.2.1.3 PRIME+PROBE

Regarding to applicability, one of the main drawbacks of FLUSH+RELOAD is that it is limited to scenarios in which the victim and the attacker share memory. This feature is not always available through deduplication in virtual environments, as most cloud providers turned it off after several attack demonstrations. However, PRIME+PROBE [70, 71] still works in virtual environments. An attacker can target the LLC cache, particularly one set of the LLC, and it will still be able to extract sensitive information. Since PRIME+PROBE is agnostic to special OS features in the system, it can be applied in virtually every system.

The original PRIME+PROBE attack was proposed for the L1 cache [13], and extending it to target the LLC is not straightforward. Since the LLC is orders of magnitude bigger than the L1 cache, it is not possible to fill the whole cache and still obtain enough granularity in the samples. However, the LLC is set-associative, so an attacker could use one of the sets to conduct the attack instead of the whole cache. The drawback of this approach is that the LLC is divided into slices, and there is an undocumented hash function that distributes the data uniformly between the slices. This slice selection algorithm makes it harder to predict in which set is going to be located an element.

As a primary step for a PRIME+PROBE attack, the attacker needs to construct an eviction set (a group of w different addresses that map to one specific set in w -way set-associative caches). Constructing an eviction set and dealing with the complex slice selection algorithm are two problems that have already been studied in the literature. As we did for the FLUSH+RELOAD attack, we first describe this requirements previous to the attack, then we describe how the attack works and continue presenting some works that have exploited this technique.

3. Related Work

Prerequisites

Various researchers have already retrieved the hash function independently. Most of them enable the use of huge pages in their systems. By default, pages of 4KB are used as we stated in section 2.1. However, other page sizes are possible (2Mb or 1GB). Such pages are known as huge pages. When the system performs the translation from virtual to physical addresses, the bits within the page remain unchanged (i.e. such bits in virtual memory will be the same as in physical memory). For example, 21 bits of the physical address will be known by the attacker if pages of 2MB are used.

In 2013 Hund et al.[72] manually retrieved the Sandybridge Hash Function. He stated that only the bits of the tag were used to compute the hash function. Later, Liu et al.[71] found that the previous statement was only true if the number of cores that the processor has is a power of 2. In that work, authors provide an algorithm to find eviction sets by using a pool of lines and iteratively adding lines to that pool and checking for conflicts between lines. Their approach creates eviction sets dynamically and does not require to retrieve the hash function. If the number of cores is not a power of 2, it is also possible to retrieve the hash function [73, 74].

Maurice et al. [4] also retrieved the hash function for multiple processors using hardware performance counters. They used processors with 2^n cores and the outcomes of their experiments are summarized in figure 3.4. Recently, Vila et al. [24] revisited the problem of finding the eviction sets, and showed that it is possible to create them without using large pages.

Address Bit		3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	0	0	0	0
		7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6
2 cores	o_0		$\oplus$	$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$		
4 cores	o_0		$\oplus$	$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$		
	o_1	$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$		$\oplus$			$\oplus$			
8 cores	o_0		$\oplus$	$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$		
	o_1	$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$		$\oplus$			$\oplus$			
	o_2	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$	$\oplus$			$\oplus$	$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$		$\oplus$		$\oplus$	$\oplus$			$\oplus$				

Figure 3.4: Complex addressing functions extracted from [4] that determine the slice of the LLC based on the physical address, depending on the number of cores of the processor

The attack

As the previously presented attacks, PRIME+PROBE consist of three steps, that are illustrated in figure 3.5: It is required a profiling stage before running this attack, in which the attacker locates the sets used by the victim. Once the set is detected, he constructs an eviction set with his own data, with memory addresses that correspond to the same cache set.

1. The attacker fills the target cache set with the data from the constructed eviction set.
2. The attacker waits for a predefined amount of time while the victim executes. As in Flush+Reload, it is possible that the attacker triggers the execution of the victim process or interrupts it.
3. The attacker reads the data that he has placed in the cache during the first step, and measures the access time. A high access time is due to that the data loaded in the first step has been evicted, with high probability, by the victim process. It means that the victim process has used the data that maps the same cache set. A lower access time means that the victim has not used that data.

Figure 3.5: Prime+Probe procedure including the three steps required to retrieve the information

This attack was first proposed for the L1 data cache in [13], and exploited by Neve et al. [75] to retrieve AES keys. This approach was later expanded to the L1 instruction cache [76, 77] and exploited to retrieve AES keys. Brumley and Hakala [78] presented cache timing templates using Hidden Markov models (HMMs) to establish a relationship between the timing observations and the cache

3. Related Work

state. All these approaches required both victim and attacker to share the same core, which is impractical.

Years later it was showed to be applicable to LLC. Although Ristenpart et al. [9] had already explored the possibility of using the LLC, the first really practical attacks against the LLC were presented by Liu et al [71] and Irazoqui et al. [79]. They both dealt with the slice selection function and targeted RSA and AES respectively. Once PRIME+PROBE was used in the LLC, it was even used to retrieve a RSA key in the Amazon EC2 cloud [80].

These attacks highly rely on precise timers to retrieve the desired information about the victim memory access. In case a defense system tries to either restrict access to the timers or to generate noise that could hide timing information, the attack is less likely to succeed. Once again, attackers have been able to overcome this difficulty. The PRIME+ABORT attack [81] exploits Intel’s implementation of Hardware Transactional Memory TSX so it does not require timers to retrieve the information. It first starts a transaction to prime the targeted set, waits and finally it may or may not receive and abort depending on whether the victim has or has not accessed this set. That is, no need for timers.

The *Invalidate and Transfer* attack [50] is somehow similar to the presented cache attacks. This attack also targets AMD processors showing that not only Intel architectures are vulnerable. It works even across CPUs that reside in the same socket but do not share the cache hierarchy. This is possible because they exploited the cache coherency protocols implemented in modern processors and multiprocessor systems.

To summarize, the attacks presented in this section target the cache, selecting one memory location that is expected to be accessed by the victim process. They consist of three stages: **initialization** (the attacker prepares the cache somehow), **waiting** (the attacker waits while the victim executes) and **recovering** (the attacker checks the state of the cache to retrieve information about the victim). They differ in the implementation, requirements and achievable resolution.

3.2.2 Transient execution attacks

Processors allow the execution of out of order instructions as long as they are committed to the machine state in the original program order. Besides, the processor uses the outcomes of the BPU to speculative execute the parts of the program that have been guessed by the BPU. In case of a misprediction or an error, these executions are reverted, and the processor discards these results. As a consequence, the program never gets to see this data. However, such executions leave a trace in the microarchitectural state that can be leveraged by an attacker to extract private information from its victim.

Figure 3.6: Logos of the two vulnerabilities: spectre and meltdown

These instructions, that the CPU executes but later discards, are known as transient instructions. Spectre [6] and meltdown [7] attacks exploit that branch misprediction and out-of-order transient instructions respectively might leak information from within the victim's memory address space without protection. They exploit this information even to break kernel isolation. These attacks rely on a side-channel as the cache (although there are other possibilities [82]), to extract the desired information, because the traces that transient executions leave on the processor are only visible at the microarchitectural level. Since the exact moment these two vulnerabilities were disclosed, this has become an active area of research.

Foreshadow [83] was presented by Van Bulck et al. as a meltdown type of attack. They were able to recover plaintexts and full cryptographic keys from Intel SGX enclaves through the cache. The extracted keys enable them to launch malicious production enclaves. Once the attack was disclosed to Intel (which named it L1 terminal fault bug), they discovered different vulnerabilities [84] which could provide an attacker with the ability to read any information from the L1 cache. This means that the System Management Mode (SMM), the Operating System's Kernel, or the Hypervisor are compromised, so are virtual machines running on public clouds.

Almost at the same time that Spectre and Meltdown were disclosed, Maisuradze and Rossow [85] studied an approach to get similar capabilities as can be obtained with spectre attacks by exploiting the Return Stack Buffers (RSBs). They demonstrated their approach by leaking keystrokes and by executing arbitrary code speculatively. These bugs have been shown to be remotely exploitable [86] and researchers have tried to hide malware in the "speculative world" [87, 88]. The list of works exploiting speculative execution is still growing [89, 90, 91, 92, 93].

Considering the attention that both industry and academia have put into these attacks, Canella et al.[94] review and identify several variants of these attacks. They systematically evaluate these attacks and some of the proposed counter-measures, uncovering new variants of the attacks and reaching the conclusion

3. Related Work

that some of the proposed countermeasures do not completely avoid the leakage.

In parallel, researchers are trying to mitigate the risks. Guarnieri et al. [95] developed Spectector, a framework that uses symbolic execution to check if a piece of code leaks more information due to speculative execution than its normal execution does. Wang et al. [96] proposed oo7, an approach that uses static binary analysis to detect potentially vulnerable binaries to spectre attacks. Schwarz et al. [97] presented ConTEXT, which requires modifications of applications, compilers, operating systems, and the hardware in exchange for protection against Transient Execution.

3.2.3 Rowhammer attacks

The Rowhammer bug is a fault attack that was discovered by Kim et al. [98]. They found out that it is possible to flip bits from a neighboring row in a DRAM without accessing it. In order to do that, they execute a sequence of memory access and *cflush* operations to force the DRAM rows to open and close frequently until one bit from another row flips. That is, the rowhammer attack takes advantage of the influence that access to a row in a particular bank of the DRAM have in adjacent rows of the same bank.

The memory location that has been altered does not have to belong to the attacker's memory region. It can belong to another process. If the adjacent rows contain critical information, this attack can be really dangerous. After this publication, many other researches studied the bug and showed different approaches. Gruss et al. [99] demonstrated an attack from a javascript extension that allows an attacker to gain access to the systems of the website visitors. They also demonstrated that the *cflush* instruction is not a requirement for the attack, since they used cache evictions instead.

The Rowhammer attack has also been demonstrated in mobile platforms [100]. They studied multiple Android devices identifying some of them that suffer from faulty devices. Xiao et al. [101] applied the attack in cloud environments to break memory isolation. Later on, Gruss et al. [102] presented another Rowhammer attack that keeps one row of the DRAM constantly open to provoke a flip in a binary, particularly in the desired opcodes. Therefore they can bypass, for example, authentication checks. They also used Intel SGX to hide the attack.

A different attack that also targets the DRAM and was introduced by Pessl et al. is called DRAMA [103]. Authors take advantage of the DRAM architecture, particularly of the shared row buffer. They first recovered the address mapping of the DRAM and later built a cross-core attack that also works in multi-processor systems.

Recently, Kwong et al. [104] presented RAMBleed. Their work shows that the Rowhammer bug can be exploited in a different way. They demonstrate that the attacker can observe bit flips in his own memory and deduce the values in nearby DRAM rows. They exploited this approach to leak an RSA-2048 key from OpenSSH, breaking the confidentiality between processes. Besides, even if the RAM has the ECC protection enabled, this attack is able to succeed as it does not require persistent bit flips.

3.2.4 Memory Bus Locking Attacks

The ring bus that connects elements in some processor sockets (e.g intel architectures), is hard to saturate and difficult to lock with normal operations. However, there are certain operations, known as exotic atomic operations, which can lock the internal memory buses. Since the processor has to ensure atomicity while processing these operations, it has to lock the memory introducing some overhead. As a difference with the previous approaches, these attacks can not be directly used to retrieve secret keys from a victim. However, they reveal interesting information about co-located virtual machines. Inci et al. [105] used this technique to discover neighbouring machines on Microsoft Azure.

Attackers can intentionally abuse hardware resources to degrade the performance of all the VMs running on the same host (Memory DoS attacks). Since these instructions have the ability to lock the memory, an attacker continuously performing these operations effectively degrades performance of real cloud applications in commercial clouds [106] and Android devices [107].

3.2.5 Other microarchitectural attacks

So far we have presented attacks against the CPU and its main components. But there are other elements included in the computer as the Graphical Processing Unit (GPU) that are vulnerable. The GPUs are traditionally used to deal with image processing tasks in the computer. However, they can be used to accelerate data intensive tasks such as the ones related to Big Data. Due to their capabilities, they are being included in data center and public clouds. In that scenario, Naghibijouybari et al. [108] demonstrated keystroke timing attacks, website fingerprinting and even retrieved the parameters of a neural network.

Intel SGX was designed to provide a trusted execution environment that only relies on hardware assuming the software can be corrupted whereas the hardware can not. However, Moghimi et al.[109] demonstrated that it not only is vulnerable to side channel attacks, but also amplifies them, effectively reducing the number of traces required to retrieve AES keys from the enclaves.

3. Related Work

The execution engine is responsible for executing the instructions in the core (Figure 2.2). Considering a simple pipeline model with three phases (fetch, decode, and execute), decode step decomposes each instruction into simpler micro-operations (uops). Each of them does a single task and the scheduler is responsible for issuing these uops to the Execution Ports. Aldaya et al.[110] presented **PortSmash**, a new attack that creates contention in these ports to gain information about other processes. They showed the feasibility of their approach against the NIST P-384 curve.

The BPU in charge of making the most likely prediction of the execution flow is also vulnerable to attacks. This unit includes a buffer, the BTB. This buffer acts like the cache, in the sense that the attacker can fill the buffer by executing a sequence of branches and decide if the victim has executed one branch by observing his own execution time. If the victim has removed one of the branches of the attacker, then his code will take longer time to execute. This idea was successfully applied to the RSA implementation of OpenSSL [111, 112]. Address Space Layout Randomization (ASLR) is a technique that randomizes the offset of key program segments in virtual memory in order to prevent attackers from deriving the addresses of such parts. Evtvyushkin et al. [113] defeated this randomization exploiting the BTB.

The PMU and the hardware performance counters intended for monitoring the performance of the software and to find bugs in the code also find a different application to perform attacks. Ushadel et al. [114] used the L1 data cache misses, L2 data cache-misses and the HPC cycle counter against AES. Some years later, the counters related to the branch predictions would be used to retrieve a RSA key of 1024 bits [115].

3.3 Countermeasures

The effectiveness of microarchitectural attacks in realistic scenarios requires appropriate countermeasures to defend against them. Some of the already proposed countermeasures suggest to avoid resource sharing to minimize the risk [54], which completely goes against the cloud paradigm. Besides, the countermeasure they proposed only works for same-core attacks. In spite of the popularity of the attacks and the fact that multiple countermeasures have been proposed in literature, to the best of our knowledge none of them are being deployed in commercial clouds. Indeed, the only precaution commercial clouds are employing is disabling shared memory pages, which mitigates the FLUSH+RELOAD attacks. However, they are still vulnerable to PRIME+PROBE attacks.

A different approach that suggests limiting the execution of the *cflush* instruction to privileged users [65] has similar drawbacks, i.e. it is only useful

for FLUSH+RELOAD. Some other proposals that suggest limiting the resolution of the time counters [116, 117] were overcome with the design of the PRIME+ABORT attack [81] or even using counting threads.

In this section we describe some of the countermeasures already suggested as did Ge et al. [118]. In order to present them, we divide them into three different categories:

3.3.1 Hardware based countermeasures

Since some of the attacks rely on either concurrent or consecutive access to shared hardware, one way to face and solve the problem is providing exclusive access to the shared hardware. Costan et al. [119] developed Sanctum pursuing the same goal as the Intel's Software Guard Extensions. A difference with Intel solution is to protect against attacks that infer private information from memory access patterns.

Wang and Lee [120] also suggested modifying the underlying hardware. Their proposal included cache designs that try to avoid collisions or to limit the amount of information that can be obtained from them. Thus, it is useful to cut down the efficacy of cache attacks or even to mitigate them.

STEALTHMEM [121] uses private virtual pages, which ensure that the data located in them is not being evicted from the cache and avoid mapping any other page with these private virtual pages (page coloring). Intel has its own mechanism known as Cache Allocation Technology (CAT) to enable system administrators to control how cores allocate data into the LLC. This technology was used in CATalyst [122] to mitigate cache attacks.

Recently, Werner et al. [123] presented SCATTERCACHE. They propose a novel cache design that uses a keyed mapping function in order to map data to cache sets taking into account different security domains. As a consequence, creating eviction sets becomes virtually impossible, which in turn, mitigates eviction-based cache attacks.

3.3.2 Detection based countermeasures

As it happens with the viruses, some researchers believe that the vacuum against side channels are side channels. That is, cache attacks exploit the side effects of running a program in certain hardware to gain information from it. Similarly, the victim can use the effects that an attacker causes to detect it, using, for example, hardware performance counters.

Since HPCs provide detailed information about the state of the processor, and

3. Related Work

are included in most modern micro-processor architectures, they have been used with defensive purposes [124, 125]. Similarly, they can be used with malicious purposes to retrieve information from co-resident user hardware utilization [126]. This may be the reason why some cloud providers disable the access to HPCs inside VMs, although some others enable this access [127].

Regarding to cache attacks, the work of Chiappetta et al. [67] demonstrated that it is possible to detect cache attacks applying different techniques, such as correlation or neural networks, to the traces that represent the attacker and the victim processes. Zhang et al. developed Clouddrader [66]. It was intended to detect attacks in cloud environments and also retrieved information from the performance counters referring to the victim and attacker VMs. Finally, Spydetector [68] is able to detect attacks based on the anomalies that the attacks cause in the victim.

FortuneTeller [128] detects a wider range of microarchitectural attacks. It uses the information collected from different performance counters in combination with advanced Recurrent Neural Networks (RNNs) algorithms to learn execution patterns from benign applications. If it finds discrepancy between the expected and the observed behavior reports a malicious behavior. Note that attacks as Meltdown [7], Spectre [6], Rowhammer [129] or ZombieLoad [91] are detected without observing them during the training phase.

3.3.3 Preemptive countermeasures

A different approach to protect sensitive applications is to specifically design them to be secure against side-channels; that is, ensuring that no memory access depend on private information and that they run in constant-time. To this end, developers can use specific tools to ensure the binary of such applications does not leak information, even if it is under attack [130] and to ensure their code does not violate the constant-time policy [131]. However, designing time-constant algorithms with key independent access patterns does not guarantee "No leakage". As the MemJam attack [132] shows, attackers can change their approach and find a way to force leakage by abusing the hardware.

CacheAudit [133] considers the information available through side-channel observations (access, timing or traces) and the binary of a program. Then it covers all possible executions of the binary in the selected system to provide formal security guarantees.

There are other tools, such as MASCAT [134], which use static code analysis techniques to detect potential attacks before running a program, as most anti-viruses do. Irazoqui et al. were able to detect implicit characteristics commonly exhibited by microarchitectural attacks. This kind of tools are effective against

malware distribution or execution. However, their effectiveness is reduced in cloud environments where the attacker does not need to infect the victim with the malware.

Wichelmann et al.[135] have developed *Microwalk*, a framework intended to help designers in the task of writing leakage free code. Their approach is based on Dynamic Binary Instrumentation and Mutual Information Analysis. They are able not only to efficiently locate and quantify memory based and control-flow based microarchitectural leakages but to accurately report to the programmer the region of the code responsible of the leakage.

CaSym [136] similarly tries to precisely identify the regions of an input program where there is leakage. This approach is based on symbolic execution and it tracks both program and cache states. It even suggests how to fix the leakage using techniques such as preloading and pinning.

3. Related Work

Reverse engineer of cache Replacement Policies

4.1 Intel cache Replacement Policies

In section 2.2.3 we stated the importance of the replacement policy in the performance of the system and we described some common replacement policies. When performing cache attacks as PRIME+PROBE that need to force the eviction of the victim's data by accessing a cache set, it is commonly assumed a pseudo LRU replacement policy. However, if we compare the performance of such attacks with the performance of FLUSH+RELOAD we observe that the second one is more precise. FLUSH+RELOAD evicts from the cache the desired line, whereas PRIME+PROBE has to trust in a sequence of access to remove the 's data. One of the reasons why PRIME+PROBE may have worst performance than FLUSH+RELOAD [71, 70], is that the required evictions are not always achieved, which in turn depends on the replacement policy that does not work exactly as expected by the attacker.

As part of this thesis, we have studied the replacement policy in recent Intel core processors. Knowing the exact eviction policy can improve the known attacks and also introduce new attack variants that may try to bypass current detection systems, as we will show in a later section (section 6.3).

It is commonly said that “most modern processors implement pseudo-LRU policies”. This assumption is based on the observations of the evictions in the cache. Regarding to Intel processors, their replacement policy is undocumented

4. Reverse engineer of cache Replacement Policies

and consequently unknown. All that is officially known is the name of the policy: “Quad-Age LRU” [137]. In this section we explain how we were able to uncover the Intel Replacement policy. Prior to the description of the experiments, we provide further information about more advanced replacement policies and prior attempts to reverse engineer the Intel replacement policies.

4.1.1 Advanced Replacement policies

Given the importance of the cache replacement policy in the general performance of the system, and the dependency of the particular access pattern of the program in the performance of the replacement policy, there have been various approaches that try to take advantage of the best features of different policies. The algorithms that are capable to adapt themselves to changes in the workloads are known as adaptive algorithms.

In 2003, Megiddo et al. [138] proposed ARC (Adaptive Replacement Cache) a hybrid of LRU and LFU. They used a parameter p to dynamically decide which item to replace. The value of the parameter p changes in response to the observed workload. One year later, Bansal et al. [139] presented their solution based on LFU and CLOCK, which they named CAR (Clock with Adaptive Replacement). It was intended as a low complexity policy that adapts itself to changes by maintaining two lists or clocks, one to deal with recency (CLOCK) and the other one with frequency (LFU).

Memory Level Parallelism (MLP) refers to the ability of the processor to have pending (to generate and service) multiple cache misses in parallel. The cost in performance is not the same for an isolated miss (more costly) than for parallel misses. For that reason, in 2006, Qureshi et al. [140] proposed to make the replacement policy aware of the cost of the misses. They proposed a new mechanism which they called Sampling Based Adaptive Replacement (SBAR) that could choose between a traditional replacement policy and a MLP-aware policy depending on their efficiency in reducing the number of cycles the processor is stalled.

In 2007 Qureshi et al. [141] suggested that performance could be improved by changing the insertion policy while maintaining the eviction policy. LRU Insertion Policy (LIP) consists in inserting each new piece of data in the LRU position whereas Bimodal Insertion Policy (BIP) most of the times places the new data in the Most Recently Used (MRU) position and sometimes inserts it in the LRU position. In order to decide which of the two policies behaves better, they proposed a Dynamic Insertion Policy (DIP). DIP chooses between LIP and BIP depending on which one incurs fewer misses.

In 2010, Jaleel et al. [142] proposed a cache replacement algorithm that makes

use of Re-Reference Interval Prediction (RRIP). By using 2 bits per cache line, RRIP predicts if a cache line is going to be re-referenced in the near future. In case of eviction, the line with the longest interval prediction will be selected. Analogously to Quereshi et al, they presented two different approaches: Static RRIP (SRRIP) which inserts each new block with an intermediate re-reference, and Bimodal RRIP (BRRIP) which inserts most blocks with a distant re-reference interval and sometimes with an intermediate re-reference interval. They also proposed using set dueling to decide which policy fits better for the running application (Dynamic RRIP (DRRIP)). Note that the policies presented in [141] and [142] were proposed by Intel.

4.1.2 Related work

The replacement policy of the Intel processors has been evolving generation after generation. According to several researchers [143, 23], Intel processors implement a pseudo-LRU policy.

The first serious attempt to reveal the cache replacement policy of different processors was made by Abel et al. [143]. In their work, they were able to uncover the replacement policy of an Intel Atom D525 processor and to infer a pseudo-LRU policy in an Intel Core 2 Duo E6300 processor. However, they could not determine the eviction policy in the other machines (Intel Core 2 Duo E6750 and E8400) they used for the experiments.

In a later work [144] they provided two different models for the Intel Core 2 Duo E6750 and E8400 processors, that seemed to match with their observations. Such policies are two different tree-based Pseudo LRU-like policies which they named PLRU-Rand and Rand-PLRU. We do not have access to machines similar to those. However, based on our experiments we believe that the L1 and L2 levels of current intel processors have inherited such legacy behavior. As a difference with their results we do not believe that the randomness they have observed really exists in these newer machines. Indeed, we have observed that when placing the data in the sets, if there is free room in the sets, the actual values of the branch do not matter. We believe that our observations on the insertion of the blocks in the cache also explain theirs. We will discuss this fact in section 4.2.

Later on, Wong [23] studied the behavior of the replacement policy in an Intel Ivy Bridge processor. He also compared its performance with a Sandy Bridge processor. For the first time, he showed that Intel processors seem to implement a dynamic insertion or eviction policy. For his experiments, he observed the time delays or latency of different access to arrays of data when they were accessed at different and constant strides. This means that the elements of an array are accessed in steps of 64B, 512B, 1KB and so on. In spite of his observations, and

4. Reverse engineer of cache Replacement Policies

since he was not using the particular knowledge of hash function and the sets, he could not provide further details about the replacement policy.

Gruss et al.[99] studied cache eviction strategies on recent Intel CPUs in order to replace the `clflush` instruction and build a remote Rowhammer attack. As they mention, their work is not strictly a reverse engineering of the replacement policy, rather they test access patterns to find the best eviction strategy. In a more recent work, concurrent with ours, Vila et al. [24] discussed how the replacement policy would affect the eviction set creation process. They also observed the adaptive behavior in Haswell processors whereas they did not observe such behavior in Skylake processors.

To the best of our knowledge, our work is the first one that provides a comprehensive description of the replacement policies implemented on modern Intel processors. We show that it is possible to predict exactly which element of the cache is going to be evicted and why.

4.1.3 Retrieval of Intel cache eviction policies

In this section our work focuses on the LLC. Since it is shared across cores, the attacks targeting the LLC are not limited to the situation in which the victim and the attacker share the same core. It is also possible to extract fine-grained information from it and many researchers are concerned about the attacks targeting the LLC. Attacks that assume a pseudo LRU eviction policy such as PRIME+PROBE or EVICT+RELOAD can benefit from the detailed knowledge of the eviction policy, as can benefit one attacker wishing to carry out a stealthy attack that does not cause cache misses on the victim. Note that low level caches (L1 and L2) can implement different policies than the LLC. In fact, they do, we will discuss it in a later section, but as for now, it is out of the scope of this section.

In order to study the eviction policy, we try to emulate the hardware in the software. We ensure that we can fill one set of the cache with our own data, access that data and force a miss when desired to observe which element of the set is evicted. Thus, we have constructed an eviction set (a group of w different addresses that map to one specific set in w -way set-associative caches) and what we call a conflicting set (a second eviction set that maps to exactly the same set and is composed of disjoint addresses). Previous works have retrieved the complex addressing function [4, 73, 74] or demonstrate how to create the aforementioned sets dynamically [71].

Although Vila et al.[24] showed that it is also possible to compute eviction sets using normal pages of 4KB, we have enabled the use of hugepages in our systems. The size of these pages is usually 2Mb. This means that 21 bits of the

physical address are known as is the mapping of the low level caches.

The eviction and conflicting set were constructed following the procedure proposed by Liu et al. in [71] (Algorithm 1). However, when the number of cores in our test systems is a power of 2, we compute the set number using the hash function in [4].

Algorithm 3 Obtaining the conflicting set

Input: *Eviction_set*, *Conflicting set candidates*

Output: *Conflicting_set*

```

function GETCONFLICTINGSET(eviction_set, candidates)
    conflicting_set  $\leftarrow$  {};
    for  $e \in$  candidates do
        read_all(eviction_set);
        flush_all(eviction_set);
        read( $e$ );
        read_all(eviction_set);
        measure time to read  $e$ ;
        if time > threshold then
            conflicting_set  $\leftarrow e$ ;
            if sizeof(conflicting_set) ==  $w$  then
                break;
            end if
        end if
    end for
    return conflicting_set

```

The procedure for obtaining the conflicting set follows the same principles and uses the eviction set that has just been constructed. This procedure is summarized in algorithm 3 and is based on the procedure proposed by Liu et al. in [71] (Algorithm 1). First, we remove all the data from one set by accessing the eviction set and then, once the data is in the cache set, *flushing* the whole eviction set. When the set is completely empty, we select one address among all the possible candidates (memory lines whose set index bits are equal to the eviction set index bits) for the conflicting set. Finally, we access the candidate address, the whole eviction set, and re-access the candidate again. If the reload time is higher than a threshold, we can conclude that the candidate address maps to the same set as the eviction set. Thus, it can be included in our conflicting set. We repeat the same procedure until the conflicting set has w lines.

Once both sets have been created, we are in condition to perform different experiments ensuring we are able to create conflicts, that is, to force evictions in the cache when desired. Note that the order of the access is important to deduce the eviction policy. It is desirable to reduce the pipelining or out of order execution effects to a minimum. We enforce this order using *mfence* instructions,

4. Reverse engineer of cache Replacement Policies

which act as barriers that ensure all preceding load and store instructions have finished before any load or store instruction that follows *mfence*.

Table 4.1: Details of the machines used in this work to retrieve their Replacement Policies

Generation	Processor	Cores	Cache size	Assoc.	OS
4th	i7-4790	4	8Mb	16	CentOS Linux 7
4th	i5-4460	4	6Mb	12	Kali Linux 2019.2
4th	i7-4770K	4	8Mb	16	Kali Linux 2019.2
4th	Xeon E3-1226	4	8Mb	16	CentOS Linux 7
5th	i3-5010U	2	3Mb	12	Ubuntu 14
5th	i5-5200U	2	3Mb	12	Kali Linux 2019.2
6th	i7-6700K	4	8Mb	16	Ubuntu 16
6th	i5-6400	4	6Mb	12	Kali Linux 2019.2
6th	i7-6567U	2	4Mb	16	Kali Linux 2019.2
7th	i5-7600K	4	6Mb	12	CentOS Linux 7
7th	i7-7700HQ	4	6Mb	12	Ubuntu 16
7th	i7-7700	4	8Mb	16	Kali Linux 2019.2
8th	i7-8650U	4	8Mb	16	Debian 9.5
8th	i5-8400	6	9Mb	12	Kali Linux 2019.2
8th	i7-8550U	4	8Mb	16	Kali Linux 2019.2

4.1.3.1 Design of the experiments

We have performed experiments in five different machines, each of them including an Intel processor from a different generation. Table 4.1 presents a summary of the machines employed in this work. It includes the processor name, its number of cores, the associativity of the cache and the OS running on each machine. We have started by studying the processor of the fourth generation, which has been a common victim of published PRIME+PROBE attacks. We have finally covered from fourth to eighth generations.

Before conducting the experiments to disclose the eviction policy implemented in each of the used machines, we have performed some experiments intended to verify that no cached data is evicted in the event of a cache miss if there is free room in the set. The procedure is quite straightforward: for each of the sets, we first completely fill it with the data on its corresponding eviction set. Next, we randomly *flush* one of these lines to ensure there is free room in the set, and we access one of the lines in the conflicting set to ensure it is not in the cache so there is going to be a cache miss. Finally, we check, by measuring times when re-accessing them, that all the lines in the eviction set (except for the one evicted) still reside in the cache. As expected, in all cases the incoming data was loaded in replacement of the *flushed* line.

Note that in a machine fully controlled by us, we can compare the actual evolution of the data in each of the sets, with its theoretical evolution defined by an eviction policy during the runtime. This is the main idea of the procedure we propose to retrieve the replacement policy. Algorithm 4 summarizes this procedure. Each of the policies that has been tested had to be manually defined. We have evaluated true LRU, Tree PLRU, CLOCK, NRU, Static and Bimodal RRIP, self-defined policies using four control bits, etc. among many other possible cache eviction policies. After multiple experiments, we can conclude that the implemented policy is the defined policy which best matches the experimental observations.

Algorithm 4 tries to emulate by software the behavior of the hardware (of the cache). For this purpose, it uses two arrays of size w . On the one hand, *address_array* mimics the studied set, storing the memory addresses whose data is in the cache set. On the other hand, *control_array* contains the control bits used for deciding which address will be evicted in case of conflict. Additionally, we need to manually define one function that updates the content of the *address_array*, one function that updates the *control_array* and another one that provides the eviction candidate i.e., it returns the address of the element that will be evicted in case of conflict. These functions are defined based on the replacement policy.

As an example, we assume we want to test the NRU policy [22], which turns out to match the policy implemented in an Intel Xeon E5620 according to our experiments. NRU uses one bit per cache line, this bit is set whenever a cache line is accessed. If setting one bit implies that all the bits of a cache set will be equal to one, then all the bits (except for the one that has just being accessed) are cleared. In case of conflict, NRU will remove from the cache one element whose control bit is equal to zero. Thus, in our procedure, the control bits would be -1 (line empty), 0 (line not recently used), and 1 (line recently used). When a memory line is accessed, the *update* function first checks if its address is already included in the *address_array*. If it is not, our function will add it to the *address_array* and set the corresponding bit in the *control_array*. On the contrary, the function only updates the *control_array*. The *getEvictionCandidate* function will return one array position whose control bit value is -1, or, if no control bit is equal to -1, one whose control bit is equal to 0. In case multiple addresses have control bits equal to -1 or to 0, the function will return the first address whose control bits are -1 or 0, that it encounters when traversing the *control_array* from the beginning. Finally, after forcing a cache miss, the *testDataEvicted()* returns the element truly evicted from the cache. We then compare it with the predicted by the NRU policy (the output of *getEvictionCandidate*).

We have noticed that only access to the LLC update the values of the control bits of the accessed element. That is, if the data is located in L1 or L2 caches when

4. Reverse engineer of cache Replacement Policies

requested (reload time lower than `ll.threshold`), we do not update the values in the `control_array`. Figure 4.1 shows the distinction between access to low and last level caches based on reload times. We have ensured that these times really refer to L1/L2 or L3 access by using performance counters. Using the PAPI library we have obtained information about misses and access to the caches whenever we have read any data and its corresponding reload time.

Figure 4.1: Distribution of the access times to different data. These times depend on which memory it was located.

4.1.3.2 Results

The outcomes of our experiments highlight some differences in the cache architecture of the machines, as also noticed in [81]. Traditionally, the cache is divided into slices, each of them containing N sets. The number of slices used to be equal to the number of physical cores a machine has. This is true for the 4th and 5th generation processors. Per contra, the newest ones have as many slices as virtual cores; that is, two times the number of physical cores. Cache sizes are similar, so they also differ in the number of sets per slice.

Since several policies suggest that different sets can perform differently, we have repeated the experiment in Algorithm 4 for each of the sets in the last level cache. As a result, we have found out that only the machines from the 4th and 5th generation implement *set dueling* to dynamically select the eviction policy with better performance between two candidate policies. We conducted several further experiments intended for determining which sets implement a fixed policy and which others change their policy based on the number of hits and misses. Locating the sets with a fixed policy is interesting for several reasons: these sets will allow to accurately determine the two different replacement policies and they will allow to favor one policy over the other depending on our interests. This also means that we could monitor one set belonging to the group of *followers* to determine which policy is currently operating.

4.1. Intel cache Replacement Policies

Algorithm 4 Test of the desired eviction policy

Input: *Eviction_set*, *Conflicting_set*

Output: Accuracy of the policy

▷ hits/trials

```

function TESTPOLICY(eviction_set, conflicting_set)
    hits = 0;
    while i ≤ num_experiments do
        j = 0, i++;
        control_array ← {}; address_array ← {};
        initialize_set();                                ▷ Fills address and control arrays
        lim = random();
        while j ≤ lim do
            mfence; j++;
            next_data = eviction_set[random()];
            measure time to read next_data;
            if time ≥ ll_threshold then                    ▷ LLC access
                update(control_array, next_data);
            end if
            conf_element = conflicting_set[random()];
            read(conf_element);                             ▷ Force miss
            candidate = getEvictionCandidate();
            if (testDataEvicted() == candidate) then
                hits++;
            end if
    return hits/num_experiments;

```

The strategies for locating the sets included different access patterns that we believed would lead to different number of misses. For example, we have simulated burst by accessing the eviction set in an ordered way, then the whole conflicting set, and finally re-accessing the eviction set. The observed number of misses depends on the policy. Pseudo LRU policies evict all the data in the eviction set after accessing the elements in the conflicting set. Whereas other policies intended for good performance in these situations (burst accesses to memory) cause fewer misses. As a result, we have located two regions composed of 64 cache sets in each slice that control each policy. Figure 4.2 represents all the sets of a cache slice with the control regions. The region coloured in blue controls the policy 1, and the region coloured in red controls the policy 2. Except for the Xeon machine, where these regions are located in sets 1024-1088 and 1280-1344, the remaining machines are consistent with the Figure 4.2.

Not all the sets within the aforementioned regions implement a fixed policy. Particularly, only one of the sets in each slice controls one policy. This fact was observed and discovered after multiple experiments with different patterns. The sets with fixed policy for each of the slices are depicted in figure 4.3. To obtain the actual control sets within the slice, it is important to test the sets without

4. Reverse engineer of cache Replacement Policies

Figure 4.2: Location of the sets controlling the eviction policy within a slice of 2048 sets. Mode 1 (blue) and mode 2 (red).

Figure 4.3: Detailed representation of the sets with fixed policy within each of the slices for the i7-4790 machine.

order, otherwise it may seem that some sets have a fixed policy and they do not.

The policy we will uncover is the one implemented in the L3 cache. We have been able to uncover a policy that seems to explain the observed evictions. In fact, over 98% of the evictions have been correctly predicted in all cases ¹, and it is likely that the errors were due to noise.

Although we have observed differences between generations and some machines implement set dueling, the decision of which data is going to be evicted is the same in all cases. The replacement policy is always the same and what changes is the insertion policy. Due to space limitations and to avoid creating confusion, we only include here the description of the policies revealed by our experiments as the ones implemented in the Intel processors. Assuming that the policy is named Quad-Age LRU, in the following we refer to ages instead of control bits. Figure 4.4 represents the procedure followed to retrieve a piece of data when requested. It summarizes the replacement policy and our observations. If the data is in the LLC, the controller decreases the age of the requested element when giving it to the processor. If there is a cache miss and one element has to be evicted, the replacement policy will select the oldest one.

Intel processors use two bits to represent the age of the elements in the cache.

¹These results refer to the sets with fixed policy in the machines that implement set dueling. The remaining sets were tested once the two policies were known and we check they followed one of them.

Figure 4.4: Diagram that represents the process of data (D) retrieval whenever the processor makes a request. The blocks with green background represent a cache hit, whereas the blocks with red background represent a cache miss.

Consequently, the maximum age is three. In the case that there are multiple blocks whose age is three, the evicted one is the first one the processor finds. The cache behaves somehow like an array of data, and when searching for a block of data placed on it, the controller always starts from the same location, which would be the equivalent to the index 0 in an array. We have observed that when all the elements in a set reach age 0, the age of all of them is increased so the processor is still able to track the accesses.

As we have already stated, the machines used in our experiments only differ in the insertion age; that is, the value that gets a cache line as age when first loaded in the set or when reloaded after a cache miss. Particularly, the machines from 4th and 5th generations that implement *set dueling*, insert the elements with age 2 in one of the cases and with age 3 in the other. We denote each of these situations or working modes as mode 1 and mode 2 respectively. The remaining machines (6th, 7th and 8th generations) always insert the blocks with age 2, which is equivalent to the mode 1 in the oldest machines.

In order to help the reader to understand how the cache works, figure 4.5 shows an example of how the contents of a cache set are updated with each access according to each policy. When the processor requests the line “d”, there is an empty block in the set, so “d” is located in that set and it gets age 2 (Mode 1) or age 3 (Mode 2). In mode 1, the eviction candidate is now “a” because it is the only one with age 3, whereas in mode 2 the eviction candidate is “d” as it has age 3 and is on the left of “a”. The processor then requests “b” so

4. Reverse engineer of cache Replacement Policies

Figure 4.5: Sequence of data access in a cache set updating their content and their associated ages for the two observed policies. Mode 1 of the 4th and 5th generations behaves exactly the same as the 6th 7th and 8th generations. The red arrow points the eviction candidate, that is, the data that would be evicted in case of cache miss.

its age decreases from 2 to 1 in both cases. Accessing "g" causes a miss. The aforementioned eviction candidates will be replaced with "g", and its age will be set to 2 or 3 respectively. Eventually, when the processor requests "a", it will cause a miss in mode 1 (it was evicted on the previous step) and a hit in mode 2, so it will decrease its age.

For an attacker, this detailed knowledge of the replacement policy implies that he can manipulate the cache at will in order to, for example, extract information about the victim usage of the cache, without even removing its data from the cache. We demonstrate this statement in a later section (section 6.3) where we present a new attack we call RELOAD+REFRESH. Besides, an attacker can prepare the cache in a way that just one access is enough to remove the desired data from the cache (section 6.4). Furthermore, attacks such as PRIME+PROBE can benefit from different access patterns that reduce the time required to evict the data from the set and to ensure that this eviction has really happened.

4.2 Intel Low Level cache Replacement Policies

As we did for the LLC, we have similarly conducted some experiments to determine the replacement policy in the low level caches. Low level caches are assumed to implement a Pseudo-LRU replacement policy, simpler than the policy implemented in the LLC. We conducted some experiments aimed to retrieve that policy.

4.2. Intel Low Level cache Replacement Policies

Based on our observations and some intuition we already had, we were similarly able to explain our results assuming a replacement policy and checking times.

It turns out that Intel implements a tree based Pseudo-LRU replacement policy in the low level caches. The tree-based replacement policy is represented in Figure 4.6. Starting from the root node it selects each of the branches depending on the intermediate values of the nodes. In the example the eviction candidate is marked in red. Since the root node contains a 1, it selects the right branch. The value of the pointed node is a 0, so it selects the right branch. Finally, the last node has a 1, so it points to the element of the right, F in the example.

Figure 4.6: Tree structure that controls the Pseudo-LRU replacement policy of L1 and L2 caches. The eviction candidate is highlighted

As we have already stated, we believe that this policy was already implemented in the Intel Core 2 Duo E6750 and E8400 processors used by Abel et al. [144]. Based on the way that each element is inserted into the LLC we conducted some experiments and checked that they are similarly placed into the lower level caches. For instance, if we look at Figure 4.6, the blocks of memory in the cache set (A to H) are ordered. According to our experiments, when the cache set is completely empty, the elements are inserted in the first free block they find regardless of the actual values of the nodes. Moreover, it seems that the initial value of all the nodes is always 0 and it does not change until the whole set is filled with data.

We have observed this behavior consistently. For example, we consider our Intel i5-7600K machine which has a L1 cache of 8 ways and a L3 cache of 12 ways. If we start an experiment with all the sets empty, and then access the whole LLC eviction set (12 elements) from A to L, if we access B after that, it is in the L1 cache. If the nodes pointing to the elements in the L1 cache set had updated with each access from the very beginning, then B should be out of the cache. This and some other tested access patterns always lead to similar results. Indeed, these observations have made us think that the randomness observed by Abel et al. [144] is more likely due to this insertions, rather than to the implementation of a random number generator in the cache control logic.

4. Reverse engineer of cache Replacement Policies

Chapter 5

Countermeasures

Section 3.3 presents some of the countermeasures that have already been proposed for cache attacks. These attacks have demonstrated to pose a great threat for confidentiality and privacy. However, to the best of our knowledge, no cloud hypervisor or OS is implementing any countermeasure. Besides, the pieces of software that carry out these microarchitectural attacks look inoffensive to most antivirus.

It was not until recently, particularly after the disclosure of the spectre and meltdown attacks, that processor manufacturers started to fix some microarchitectural bugs. They tried to avoid the root cause of the leakage (speculative and out of order execution). However, the channels that allow to extract the private information of the users, the side-channels, are still there. There was a lot of hype around spectre and meltdown when they were publicly revealed. They also raised concerns about the security of the hardware in the general opinion and about the performance degradation that may cost fixing them.

From the attacker's perspective, the advantages of performing these microarchitectural attacks are clear. They are stealthy to most antivirus, they allow the extraction of private data from the victims and they mainly rely on the hardware. In addition, if the attacks are performed in a cloud environment where multiple VMs share the same infrastructure, the attacker does not even need to gain access to the victim machine or the hypervisor. Sharing the hardware is just enough to gain information about the victim process.

On the other hand, from the user's perspective, there are few things he can do to protect himself from this threat. As stated before, hardware countermeasures may take years to deploy and they are not implemented. Some other countermeasures (code analysis) are intended for the developers of the cryptographic libraries, in which end users have to put their trust. A different approach, to

5. Countermeasures

detect the malicious code before its execution, only works if the attacker tries to distribute his code. Finally, the proposed detection systems need to correlate and get information from the attacker and from the victim at the same time. Apart from the overhead that this approach introduces in the system, an end user may not have enough permissions to do so.

Figure 5.1: Diagram of the existing layers between the applications and the hardware. The OS provides isolation between processes in the non-virtualized environment whereas the hypervisor provides logical isolation between VMs.

This chapter evaluates two different detection approaches: cache-attack detection based on timing measurements and detection based on performance counters. Additionally, the second approach is evaluated to detect other microarchitectural attacks rather than the cache-attacks. The approach is similar in all the considered scenarios, using the same information available for a user of a probably vulnerable cryptographic library. The considered scenarios (virtualized and non-virtualized) are depicted in Figure 5.1 with their respective layers.

5.1 Detection of cache attacks using execution time

Cache attacks mainly use the timing information leaked from the cache to derive private information from their victims. This timing information derives, in turn, from the memory hierarchy; particularly, it derives from the noticeable time differences between *cache hits* and *cache misses* (loads from the DRAM). The times are usually measured using the `rdtsc` instruction, which is not privileged, so it can be used by the victim to monitor his own processes.

The `rdtsc` instruction is present in all x86 processors since Pentium. It reads the time-stamp counter from the processor and places that data into the EDX:EAX registers. This counter contains the amount of clock cycles since the last processor reset. For further details the Intel Software Developer’s Manual provides more detailed information: Volume 3, Chapter 15 [28]. This instruction

is the key to perform a timing characterization of a process executing a cryptographic algorithm: AES. The same AES encryption process is characterized when performing normally, when suffering a cache attack, and when sharing the CPU with other different processes, to evaluate how they affect to it. The main objective is to get accurate timing models that allow the detection of the attacks.

This approach is particularly interesting in cloud environments. First, in cloud environments it is more likely that victims and attackers share a physical infrastructure, and cloud providers do not currently provide protection mechanisms against these attacks. Second, the information retrieved from `rdtsc` is available inside the VM. For these reasons, the characterization of the encryption process running inside a VM is interesting.

5.1.1 Case of Study: AES

The T-table implementation of AES, briefly described in Section 2.7.1, is vulnerable to cache attacks because its access to memory (to the tables) depends on the key values. In addition, not all the lines of the cache holding the table are accessed during the encryption process. Indeed, the probability of not accessing a concrete cache line within an encryption is given by Eq. 5.1, where n represents the number of table entries a cache line can hold, and N_r the number of rounds of the algorithm.

$$Pr[no\ access\ Te_i] = \left(1 - \frac{n}{256}\right)^{N_r * 4} \quad (5.1)$$

Each table Te is accessed 4 times each round. n depends on the size of the cache line (B) and represents the number of entries of the table that can be monitored by observing a cache line. Since each value in Te has 32-bits (4 bytes), $n = B/4$. Considering AES-128 ($N_r = 10$) and a cache line of 64 bytes ($n = 16$), the probability of not accessing the cache line during the encryption process is 7,5%.

5.1.1.1 Modeling AES

The modeled process is a simplified version of a server. It receives requests that are processed; i.e. an encryption of a block of data is performed whenever a request is received. This server could be one of the represented applications of Figure 5.1. It can receive legitimate requests or tricky requests, when an attacker is manipulating the cache before sending the request. FLUSH+RELOAD, as described in [31], is the technique used to perform the attack:

- Flush one or more lines of the cache containing some known data from the T-tables.
- Send a request to the server, and wait until it performs the encryption.

5. Countermeasures

- Reload the removed data again, measuring the elapsed time.

As stated before, time measurements are done using the *rdtsc* instruction. Low reload times mean the data has been recently used (cache hit) whereas high reload times mean the data was not used (cache miss). Figure 5.2 represents a distribution of reload times when performing an attack flushing four cache lines, one for each T-table. It is easy to see which access correspond to cache access (even between cache levels) and which ones correspond to cache misses or main memory access. Note that these times depend on the hardware. They also give an idea of the time penalty that a process suffer per cache miss. Times over an established threshold, at 200 cycles in this example, represent cache misses.

Figure 5.2: Histogram of times required to read (Reload) a cache line when performing an attack against AES including 40 million of samples.

The histogram represented in Figure 5.2 is coherent with the theoretical probabilities that can be estimated using the equation 5.1. The values to the left of the threshold, associated to cache hits at different levels of cache, represent more than 90% of the total samples. The probability distribution between the different cache levels depends on the attacker and victim sharing the same core (sharing L1 cache) or not (sharing L3 cache).

In order to detect cache attacks, first it is required to gain knowledge about how an encryption process should behave under different circumstances. In order to obtain a timing model as accurate as possible several experiments are needed. In the server application, inserting the *rdtsc* instruction before and after performing an encryption provides the encryption time. In order for an attack to be successful and obtain the full key, it is mandatory to get information of the four T-tables. However, in order to perform a successful attack, the attacker does not have to flush four lines each time an encryption is performed. As representatives for detection, this work considers the cases in which the attacker flushes from one to four lines per encryption.

5.1. Detection of cache attacks using execution time

The experiments are performed using the AES T-table implementation of OpenSSL, version 1.0.1f, in a machine with an Intel Core i7-4790(3.60 GHz) processor with Centos7 OS. The experiments in a virtualized environment are performed in an Intel Xeon CPU E3-1226 v3 (3.30GHz) also running Centos, belonging to an openstack cluster with KVM as hypervisor and the deduplication feature enabled. Both machines have a L3 cache of 8MB with a line size of 64 bytes. Each of the considered experiments contains up to 1 million encryptions.

Figure 5.3 depicts the distribution of time elapsed in different cases for the i7 processor: when performing no attacks and when flushing different number of cache lines, from 1 to 4. The encryption process lasts around 600 cycles, during each of the ten rounds data is accessed and retrieved from the L1 cache, which needs less than 50 cycles. Although each round each table is accessed 4 times, the processor performs several reads “simultaneously” because of the pipeline. Also, the distance between peaks in Figure 5.3 is consistent with the main memory access time (*cache miss*). This work characterizes the FLUSH+RELOAD attack, although other attacks as PRIME+PROBE are expected to behave similarly, as they evict the lines the victim is about to use from the cache.

Figure 5.3: Distribution of times required for encryptions under different circumstances (attacks flushing different number of lines per encryption or normal operation) for the experiments performed in the Intel Core i7-4790. We use 1 million samples per distribution.

The same experiments are performed between two different co-resident VMs. Figure 5.4 depicts the elapsed time distribution obtained. In this case, the normal encryption process takes around 1000 cycles. That is, the time it takes to execute the same process varies between scenarios. At first sight, the main conclusion drawn from Figure 5.3 is that, as long as there is no other process executing in the CPU, times above 700 cycles are highly suspicious of being causing an intended delay. Whereas if the process is running inside a VM, this threshold is about 1100 cycles. Besides, the peaks of the time distributions in cloud environments are wider than those of non-virtualized processes; distinguishing between

5. Countermeasures

Figure 5.4: Time distribution of encryption in our cloud environment under different circumstances (attacks flushing different number of lines per encryption or normal operation) for the experiments performed in the Intel Xeon E3-1226. We use 1 million samples per distribution.

distributions is much harder, specially when several lines are flushed. Even some non-attack encryption times are above this threshold.

Note that these distributions are not only dependent on the machine and the scenario but also on the frequency at which the processor is running. When collecting the samples, the server was continuously answering requests. Since AES is a high CPU consuming process and our governor adjusts the frequency of the processor at its maximum, we consider the frequency as a constant.

Table 5.1: Probability distribution of cache misses (%)

N	Amount of Cache misses (penalty)									
	0		1		2		3		4	
	T	E	T	E	T	E	T	E	T	E
1	7.6	8.6	92.4	85.9	0.0	1.0	0.0	1.0	0.0	3.4
2	0.6	0.8	26.7	32.6	72.7	59.5	0.0	1.5	0.0	5.6
3	0.0	0.0	6.7	10.4	45.3	51.3	48.0	31.9	0.0	6.4
4	0.0	0.0	1.7	3.2	21.3	32.6	50.9	44.9	26.1	19.3

The distribution obtained in the non-virtualized environment is used to compare the theoretical analysis with the experimental results. The theoretical analysis is performed using the formula given in equation 5.1. The experimental results depend on the selection of thresholds to distinguish between peaks. The first peak (the one corresponding with the normal operation) assumes that there were no misses during the encryption. The second one, represents one cache miss, and so on and so forth. The probability is then computed as the (Number of samples in each range)/(Number of total samples). Table 5.1 shows the probability of cache misses (columns) depending on the number of cache lines flushed (rows). Column T shows the theoretical estimation and column E the experimental result.

The theoretical and the experimental distributions do not match exactly. However, using a metric to compute how similar the distributions are, as the Kullback–Leibler divergence, is enough to decide which distribution matches with which situation. Thus, in an ideal scenario, it is possible to accurately distinguish if an AES encryption process is suffering an attack or not.

5.1.2 Noise tolerance

In the real world, a process performing an encryption is likely to share the CPU with other processes. This fact is more remarkable in cloud environments, considering each VM share computing resources with many other VMs in order to achieve higher CPU utilizations and economies of scale. The following experiments are performed to model how sharing resources affects the encryption process, with and without an attacking process, to refine the proposed timing model. The goal is to be able to detect the attacks under different conditions. The experiments consider three different kinds of interfering tasks: CPU load, memory load and database management systems.

The first approach consists in launching the *Lookbusy* program and, at the same time, perform the encryptions with and without the attack. *Lookbusy*¹ stresses, with a synthetic load, different CPU hardware threads to a certain utilization avoiding memory or disk usage. The experimental setup includes different utilization levels. The attack is able to succeed with a similar number of samples in this scenario. However, the histograms of the times are greatly influenced by the CPU load.

Figure 5.5: Encryption time histograms with different CPU utilization levels, for the process running normally in the i7 processor.

Figure 5.5 shows the effect of sharing the CPU with other CPU consuming

¹<http://www.devin.com/lookbusy/>

5. Countermeasures

processes. As the use of CPU increases, the height of the peak representing “non attack” decreases, and other peaks rise up. At some point, the “non attack” distribution begins to look like the attack distribution when one line is flushed. Figure 5.6 is a similar diagram that depicts the distributions separately for the different loads as bar graphs, in order to observe this effect clearly. It shows an interesting effect: the height of the “non attack” peak seems to be linearly related to the CPU utilization. Seeing this fact from a different point of view, there is an information leak about physical CPU utilization which is exploitable by observing and analyzing our own execution times of a modeled process.

Figure 5.6: Distribution of times around peaks for different CPU utilization levels. The process was running normally in the i7 processor.

The second approach is to consider the use of memory as a possible interference. *RandMem*² is an access memory benchmark. The experimental setup includes an adapted version of *RandMem* to access random memory regions individually. Each experiment executes *RandMem*, accessing memory regions using a random pattern, simultaneously to the AES encryptions. The experiments are repeated with different memory size to stress, measuring encryption time under these circumstances. As a difference with *Lookbusy*, the execution of *RandMem* also consumes CPU and not only memory. Figure 5.7 represents the effect of memory consumption on the encryption times.

It is important to check if the measured delays are due to cache misses or not in memory consuming processes. Consequently, as performance counters are available for non-virtualized processes, the number of cache misses per encryption is available. It turns out that the memory consuming processes are only able to cause at maximum one cache miss per encryption and less than 1% of the

²<http://www.roylongbottom.org.uk>

Figure 5.7: Encryption time distributions with different memory utilization levels. It includes 1 million samples of the normal execution of the encryption per distribution in the i7 processor

encryptions are affected. That is, even when the system is globally using lots of memory, the interference in cache between the encryption process and the memory consuming processes is insignificant unless intentionally caused for the AES encryption process by an attacker.

The main observable difference between Figures 5.5, with CPU intensive loads, and 5.7, with access memory intensive loads, is that the latter is barely affected for some memory intensive workloads. When it is affected, it is highly probable that the CPU utilization generated by the benchmark is causing the deviation. Thus, the contribution of the CPU consumption is much more significant on the execution time distribution.

Although the reported results refer to the physical machine with no virtualization, experiments in cloud environments show a similar distribution. The third approach considers an additional workload for cloud environments that mimics cloud data serving systems. To this end, the Yahoo! Cloud Serving Benchmark (YCSB) is selected to coexist with the AES encryption process. The YCSB is used to simulate workloads that fetch web pages, including the writing portions of those workloads [145], as these are realistic loads in cloud. Again, the outcomes of the experiments can be seen in Figure 5.8, altogether with the peaks distributions in Figure 5.9. Note that 0 VM means that only the server and the attacker are running in the host.

Considering virtual environments, it is also possible to approximately determine the CPU usage of the physical host in which the process is being executed. Therefore, this scenario includes in the experimental setup measuring the host machine utilization. Each launched instance consumes about 25% of the CPU. With 3 instances launched to set CPU usage context and two more VMs, used as server and attacker in the experiments, the CPU utilization is about 100%. With

5. Countermeasures

Figure 5.8: Encryption time distributions of the normal encryption with co-resident virtual machines running the YCSB benchmark in the Xeon E3 machine.

Figure 5.9: Distribution of times around peaks representing the normal encryption process with co-resident virtual machines running the YCSB benchmark in the Xeon E3 machine.

more than three running instances, CPU utilization is about 100% and it slightly decreases when executing the attack simulation processes. In terms of timings, there are no significant differences between the different types of instances; the most relevant parameter is CPU consumption.

Other important fact that arises from this experiment is that all the distributions are similarly affected by the noise. For example, in virtualized environments and with a 100% CPU usage, the distributions for each of the considered conditions are still distinguishable. Figure 5.10 shows the time distributions for the different conditions: under no attack or flushing from 1 to 4 cache lines when the CPU utilization of the host is about 100%. The distributions have moved to the right, increasing the time required to perform each encryption. This shift can be easily observed by comparing Figure 5.4, for the no-load scenario, with the

newly generated Figure 5.10. The shapes of the distributions are similar.

Figure 5.10: Distribution of times for encryptions in virtualized environments 100% CPU usage.

5.1.3 Proposed Detection Algorithm

Based on the behavioral models presented in the previous section, this work provides a simple detection algorithm summarized in algorithm 5. The proposed algorithm decides whether the system is under attack or not based on the time distribution measured. The idea behind this algorithm is not to obtain the most accurate detection algorithm, but to show that even the timing information is useful to detect cache attacks and that even the simplest detection algorithm performs reasonably well.

The algorithm includes three main steps: computation of the encryption times distribution and location of the peaks, analysis and decision. In order to compute the distributions, this detection algorithm uses a window including 200 samples. That is, we store the encryption times of the last 200 operations. Then, the algorithm computes an histogram with the windowed samples selecting time intervals of 20 cycles (a compromise between the number of slots and accuracy). Once the histogram has been calculated, it is used to detect where the peaks of the distribution are located. We use a sliding window of three bins, i.e. it contains the previous, current and next samples, that is moved through the x-axis of the distribution. Finally, we use this information to locate the peaks of the distribution.

Note that with this approach the algorithm does not really need to use the information about where the peak of the normal distribution should be. It is general and it does not take as an input a model for each machine or scenario. Although a more specific model for each case will have performed better, our goal was to show that it works.

5. Countermeasures

Algorithm 5 Cache attack detection algorithm

Input: Encryption times, confidence level

Output: *flag detected*

```
h = calculate_hist (timesn-200...n);  
w = window (h);  
arr_peaks = peak_search (w);  
if length (arr_peaks) > 1 then  
    if arr_peaks [0] < arr_peaks [1] then  
        confidence ++;  
    else  
        if (num_significant_peaks ≠ p-0) >= 2 then  
            confidence ++;  
        else  
            confidence --;  
        end if  
    end if  
else  
    confidence --;  
end if  
if confidence > threshold then  
    detected = 1;  
end if  
return confidence, detected;
```

If more than one peak is detected, the heights of the peaks decide whether the encryption process is attacked or not. For example, according to the previous models, if the second peak is higher than the first one this process is detected as an attack. No further considerations are taken in this case than checking if the first peak is over a certain value or checking if the two peaks are in fact the same by looking where the peaks are placed. This distribution is similar to the distributions with more than 50% of the CPU being utilized. The reported utilization in real cloud environments such as the Amazon EC2 cloud, is below 30% at maximum [146]. Therefore, the assumption should not be a problem.

Similarly, there are almost no situations in which the normal distribution has more than two peaks, even in noisy environments. As in the previous case, if the number of peaks of the sampled execution includes more than 2 peaks, the algorithm considers that there is an attack going on.

However, some false positives should appear. In order to reduce the number of false positives, this work includes a new parameter named “confidence”. If the new samples with their respective previous samples within the window are repeatedly classified as an attack, the confidence increases until it reaches a threshold. Then the algorithm definitely triggers an alarm to notify the user that something

5.1. Detection of cache attacks using execution time

is going on. On the contrary, the value of the confidence decreases.

A test set, different from the training set used in the previous section to model and implement the algorithm, is required to validate the detection system. The test set includes simulations of multiple encryptions with different conditions: under attacks on different number of lines, under no attack in both virtualized and non-virtualized environments and including loads below 30% of the CPU utilization. The algorithm is applied in runtime to detect the attacks.

A false positive is a non-attacking process labelled as an “attack”. On the other hand, a false negative is an attack that is able to succeed, not being detected in time. The rates of false positives and false negatives are shown in tables 5.2 and 5.3 for different number of lines flushed, where 0 lines flushed means no attack. Most of the published versions of the attack against the T-Table implementation require around 100000 samples to succeed or even more. However, this Thesis include a proposal in Section 6.2) that obtains the key with only around 3000 samples in non-virtualized environments and with about 10000 samples in virtualized environments when monitoring four lines at a time. For these reasons, these lower values are used to evaluate the proposed detection system.

Table 5.2: False positives rates(%) including 1000 request per experiment.

Flushed lines	Non-virtual environment	Virtual environment
0	4%	4.8%

Table 5.3: False negative rates(%) computed with 1000 request per experiment.

Flushed lines	Non-virtual environment	Virtual environment
1	2%	3.6%
2	1.1%	2%
3	0.5%	2%
4	0.4%	1.5%

Even though the proposed detection algorithm is quite simple, it is able to detect over 96% of the attacks. However, the performed experiments have shown that, for higher CPU utilization, false positive rates can increase as the distributions are more similar to the attacking ones. This fact is not properly considered in this simple algorithm. For instance, a more advanced approach could use a test process to infer the CPU utilization of the host and adapt the models accordingly. Moreover, in our studied scenario, there are more false positives when the encryption process has just begun and the times have not yet stabilized, in transient periods. This time corresponds to the period of changing frequency in the processor. If there are no processes running, the governor sets this frequency to the minimum. When a process starts its execution this frequency rises up to the maximum. As a result the algorithm could be similarly improved by using this

5. Countermeasures

frequency information. Note that for a different process or a different governor the frequency may not rise up to the maximum.

The proposed algorithm provides a proof of the utility of the models obtained. But there is still room for improvement. Besides adding information about the CPU utilization or the frequency of the processor, improvements on the peak extraction algorithm are also possible. For example, during the experiments the same peak is sometimes labeled as two different peaks. This could be avoided increasing the window size. Furthermore, machine learning techniques, such as neural networks or predictive algorithms, could be applied to the models to perform detection. The drawback of this approach is that each model would be machine dependent, which diminishes practicability.

When an attack is detected, stopping the encryptions provides information to the attacker, who can react and escape. Besides if an attacker observes this behavior in a system, it is likely to keep trying so the server does not work properly, performing a Denial of Service (DoS) attack. However, if a victim that detects the attack keeps on performing encryptions with a fake key, it increases the chances to identify the process responsible of the attack, considering that the permission level is adequate. In contrast, in cloud environments, a user is not able to kill other clients machines, or even to decide where her machine is placed. The fake key is useful to keep the attacker busy and away from the real key while notifying the cloud provider that there is an attack from a co-resident VM. It is in hands of the cloud provider to detect the attacking VM. Even if the victim has information on a suspicious VM, that can be attacking as well other VMs from other cloud clients, certainty is required in order not to defame other clients. However, using a fake encryption key, the attacker is expected to stop the attack once it believes it to succeed. The victim can continue working normally once it detects the attack has finished, with the risk of suffering a DoS if the attack never stops.

5.2 Detection of cache attacks using hardware performance counters

Modern computing technologies like cloud computing build on shared hardware resources opaquely accessible by independent tenants, ensuring protection through sandboxing techniques. However, although this isolation is solid at the logical level, ensuring that tenants cannot access each others memory, hypervisors cannot properly prevent information leakage stemming from the shared hardware resources such as caches. This previous statement also applies to OS, which similarly provide logical, but not physical, isolation between processes. Compared to

5.2. Detection of cache attacks using hardware performance counters

other resources like BPU or the DRAM, caches can be exploited to recover fine grain information from co-resident tenants in shared environments.

Most of the proposed mitigation techniques succeed in stopping cache based attacks, but are not being adopted by Cloud Service Providers (CSPs). Proposed hardware countermeasures require making modifications to the hardware that not only induce severe performance penalties but also take years to integrate and deploy into the infrastructure. Cloud hypervisors, on the contrary, can implement any of the proposed hypervisor based countermeasures [121, 147, 148] by just making small modifications to the kernel configuration. Despite the immediate fix that these countermeasures would provide, they are not being adopted by cloud providers, mainly due to the constant performance overhead that they add to their systems. Other feasible mitigation proposals consider periodic VM migration to avoid long-term co-location. VM migration, however, also introduces extra overhead whether there is an attack or not. Other proposals suggest that just as the attacker uses a side-channel to obtain information, the VM can defend itself by using a side-channel to detect co-resident tenants with possibly malicious intentions [149]

The current situation leaves tenants with little help from hardware and hypervisor designers or cloud service providers to protect themselves against cache attacks. The situation is similar as the one observed with the detection of cache attacks based on times, except for a few things. One of the most important changes is that major hypervisors have virtualized hardware performance counters and are consequently accessible within the VMs if properly configured. The previous approach, that showed to be useful for AES, has one major drawback for RSA: an attacker in an ideal scenario can succeed just monitoring one single RSA decryption [14]. This means that, by the time the victim gets just one single sample for the attack detection, the attack can have already retrieved the key. Plus, the timing approach requires different models for the different machines and algorithms, and consequently some knowledge and previous work from the users.

Thus, there is the necessity of giving those tenants or applications that *voluntarily* want to protect against cache attacks tools to defend themselves. So far, all known cache attacks have in common that they cause cache misses *in the victim VM process*. Thus, detecting an anomaly in the number of cache misses in the victim can indicate an ongoing cache attack and, thus, trigger VM migration or other actions to mitigate the attack.

Cache misses can be obtained by reading the hardware performance counters found in all modern processors. These hardware event counters track hardware events such as cache misses, and were originally intended to enable the detection of bottlenecks in executed software. Optimization is not the only application of these counters, it has also been demonstrated that the hardware performance

5. Countermeasures

counters are also useful to detect malware and security breaches [125, 124, 150, 151]. Libraries such as PAPI, facilitate the task of configuring and reading the hardware counters.

There have been several attempts to detect cache attacks using the hardware counters [67, 152, 66], but they have strong drawbacks. Some works require the hypervisor to periodically monitor all existing processes, which introduces a great overhead in CPU usage and depends on how efficiently an attacker can hide from the monitoring tool [66, 152]. Other works offer solutions applicable to multi-process environments, but not feasible in cloud environments [67, 65].

This work propose using a monitoring service, that detects anomalies in the cache miss hardware performance counter, inside the VM *only in the victim side*. The monitoring service can be activated on demand inside the VM; this way, it is the user of the tool who decides if it is worth it the minimum performance penalty in exchange of improved security. Moreover, its VM is also the only one affected by this penalty. As previously mentioned, the hardware performance counters must be exposed to the VM in order to be feasible. Just changing the configuration of the hypervisor, it is possible to enable performance counters access inside the VM. This access can even be enabled only in the VMs that request the service. As the hypervisor is responsible for the virtualization of the counters that can be read inside a VM, they refer uniquely to this VM. That is, one VM can not read counters referring to another VM, even when they share the hardware. Right now, most cloud service providers only expose them if the customer is renting the whole machine, probably due to their fear of exploitation as a side channel in hardware shared by various tenants. However, current attacking techniques exploiting shared hardware expose much more information than hardware counters would.

This work demonstrates, for the first time, that hardware performance counter access for tenant VMs can indeed be utilized to *improve security* of the tenants. A new monitoring service, CacheShield, is offered to tenant VMs to detect cache attacks. CacheShield can be activated before running sensitive processes. CacheShield detects attacks quickly and with high reliability and low CPU overhead, due to the use of Page’s cumulative sum method [153]. The CUSUM method is an unsupervised anomaly detection method, ensuring that even new attack techniques can be detected with high confidence. CacheShield automatically turns off if the monitored process is idle by detecting the lack of activity, resulting in a significant reduction in CPU processing overheads.

The objective is to build an attack detection tool that detects any abuse of the LLC without any modifications to the hypervisor, OS, or the CPU hardware. To summarize, the proposed tool:

- builds a performance counter based monitoring service that users can vol-

5.2. Detection of cache attacks using hardware performance counters

untarily activate to detect when they are under attack.

- is easy to use even by users with limited knowledge of the attacks.
- monitors uniquely the victim process upon when active, i.e., the cloud service provider does not waste cycles continuously monitoring all processes.
- does not need any help from the underlying system (only requires the hypervisor to enable VM access to the performance counters, a feature commonly supported by all major hypervisor systems, including KVM, VMware and Xen).
- implements an efficient algorithm that maximizes fast and reliable attack detection, while minimizing false positives and keeping the performance overhead minimal and restricted to the victim VM.
- succeeds detecting all existing cache attacks, including stealthy attacks that are miss-detected by other solutions, e.g. FLUSH+FLUSH since the detection algorithm uses attack characteristics that are independent of attack and victim behaviors.

5.2.1 Performance counters

HPCs were introduced in Section 2.6. The list of available events focuses on waiting periods (e.g. clock cycles the processor is stalled), memory or bus access (e.g. cache misses or DRAM requests), and other performance-critical metrics like branch prediction or TLB events. Cloud providers might disable the utilization of performance counters from guest VMs. There are two main reasons why they would do this:

- Performance counters might be utilized with malicious purposes, similarly to the way the thermal sensor was used in [154], and retrieve information from co-resident user hardware utilization [126], which in theory should not be possible as the hypervisor only gives information to each VM about itself.
- As performance counters are hardware dependent, giving a guest VM access to benign utilization of performance counters might be problematic if guest VMs are migrated over different architectures, as customers would have to design code for different hardware architectures.

However, these facts should not make cloud providers disable the usage of performance counters from guest VMs, specially when one can use them as a protection mechanism. In fact, attackers have already found alternative ways to

5. Countermeasures

retrieve the same information performance counters give. For instance, attackers can read the cycle counter or an incremental thread to know when TLB or cache misses occur. Thus, disabling the counters does not entirely prevent the leakage of hardware events information. As for the second claim, a possible solution could be to create clusters with the same hardware configuration, and migrate VMs within this cluster. Thus, if access to the guest VM performance counter offers more protection to cloud infrastructure customers, these arguments are not strong enough to cancel it.

5.2.2 Detection, mitigation and other countermeasures

HPCs have been used to detect generic malware [151, 150, 155] as well as microarchitectural attacks [67, 66, 65, 152]. Their success mostly depends on the ability to correctly identify cache (and other resource) attack patterns monitoring the associated event in the HPC. This approach is usually implemented at the OS or hypervisor level, that has enough permissions to monitor what is running in the system. However, there are two main problems with these detection-based approaches:

- Most of these detection approaches incur severe performance overheads that hypervisors or OSs do not seem willing to pay. No OS seems to be implementing such a mechanism. This leaves the user of the system with few resources to know whether her code will be executed in a safe environment.
- As these detection countermeasures base their success on the monitoring of both the victim and the attacker processes, the attacker can vary patterns in a smart way to try to bypass the detection mechanisms.

These facts are observed, for instance in [66, 152, 67]. All three works incur significant overheads on all applications. CloudRadar, for example, requires three dedicated cores for its detection [66]. In addition, they usually assume the ability to monitor the attacking process [65, 67], which is not possible across VM boundaries (except for the hypervisor), and usually not even possible for user-level processes. As a consequence, cloud providers may not want to implement such a tool, as it increases overall system cost, while the benefit of preventing cache attacks might be a benefit only few customers are willing to pay for. Indeed, when this detection algorithm was proposed, there were no CSPs employing VM monitoring for microarchitectural attacks.

Regarding to the second statement, FLUSH+FLUSH was specifically designed to bypass such detection systems. Indeed, taking into account the counters that they use for discovering the attack and the nature of FLUSH+FLUSH it seems

to be undetectable to these systems. Indeed, FLUSH+FLUSH is one of the main motivations of this PhD Thesis to develop CacheShield. At the time CacheShield was designed, the PRIME+ABORT attack was published, which includes a *prime* step that evicts the data from the cache. This approach detects this previously unknown cache attack.

5.2.3 Victim-based Attack Detection

In order to detect an attack it is important to first analyze the behavior of the *victim applications* by monitoring several critical hardware performance counters. This behavior of critical applications is analyzed in the presence and absence of various cache attacks, and further analysis is performed to determine how well each counter serves as an indicator for ongoing attacks.

For the sake of simplicity, the analysis is based on cryptographic algorithms, which are the most popular target for cache attacks: the implementations of AES and RSA introduced in Sections 2.7.1 and 2.7.2: the software AES T-Table implementation and the RSA sliding window implementation (with flag `RSA_FLAG_NO_CONSTTIME` set) of OpenSSL 1.0.1f, which give representative results for public key and symmetric key cryptography. However, the proposed approach can also detect attacks on other security-critical pieces of code like SSL/TLS protocol stacks.

The FLUSH+RELOAD against both implementations is used as sample attack. FLUSH+RELOAD tries to gain information from the execution of certain instructions or from the access to certain data which depend on the key. For the used version of RSA, attacks target the instructions (depending on the implementation RSA can also be attacked considering access to data), while AES attacks are an example of cache attacks focused on the data.

AES has been used in the previous proposed countermeasure, and the effect of the attack on timing has been proved. Figure 5.11 presents the histogram of times measured in cycles for the RSA decryption process. Similar to AES, the encryption time greatly increases when suffering an attack. But execution time is not only machine and algorithm dependent, as this figure shows, but also key dependent. In turn, this means the exponentiation algorithm is not constant time, and this timing difference can be exploited to leak information about the secret key being processed by this algorithm. Note that, in the case of the sliding window exponentiation, both the operations and the access to data can be exploited.

The following sections include an analysis of the hardware events which are relevant for the detection system, a description of the implemented detection algorithm and an evaluation of this performance counter based approach.

Figure 5.11: Histogram of the number of cycles measured for the decryption process of RSA. From the OpenSSL sliding window implementation and a key length of 2048 bits. It includes 1 million samples and different keys

5.2.3.1 Analyzing Hardware Performance Events

Modern server CPUs make a large number hardware performance counters available, but only a limited number, typically 4 to 8, can be monitored in parallel. The Performance API (PAPI) [29] is used to access the performance counters. PAPI provides sufficient resolution to detect attacks while it also simplifies the task of collecting performance data. In this preliminary step, data from 30 accessible hardware event counters is collected on the test platform, for sample public (RSA) and private (AES) key algorithms, in the presence and the absence of cache attacks.

The PAPI interface provides instructions to read the counters referring to the encryption process before and after each cryptographic operation; that is, a detailed information about the variation of the counters for a single encryption or decryption execution is available. The idea is similar to the previous one of measuring the times per execution of the algorithm [156]. Since the number of counters that can be read at the same time is limited, data is collected for different groups of counters at different times. Data is joined afterwards to compute the statistics to determine and quantify which counters provide meaningful information to detect the attacks. Note that at this point, the collected measurements refer to a single execution of each cryptographic process, when working normally and when suffering an attack. The complete list of all the events that have been considered includes:

PAPI.L1.DCM	Level 1 data cache misses
PAPI.L1.ICM	Level 1 instruction cache misses

5.2. Detection of cache attacks using hardware performance counters

PAPILL2.DCM	Level 2 data cache misses
PAPILL2.ICM	Level 2 instruction cache misses
PAPILL1.TCM	Level 1 cache misses
PAPILL2.TCM	Level 2 cache misses
PAPILL3.TCM	Level 3 cache misses
PAPILCA_SNP	Requests for a snoop
PAPILCA_SHR	Requests for exclusive access to shared cache line
PAPILCA_CLN	Requests for exclusive access to clean cache line
PAPILCA_INV	Requests for cache line invalidation
PAPILCA_ITV	Requests for cache line intervention
PAPILL3.LDM	Level 3 load misses
PAPILTLB_DM	Data translation lookaside buffer misses
PAPILTLB_IM	Instruction translation lookaside buffer misses
PAPILL1.LDM	Level 1 load misses
PAPILL1.STM	Level 1 store misses
PAPILL2.LDM	Level 2 load misses
PAPILL2.STM	Level 2 store misses
PAPILPRF_DM	Data prefetch cache misses
PAPIMEM_WCY	Cycles Stalled Waiting for memory writes
PAPILSTL_ICY	Cycles with no instruction issue
PAPILFUL_ICY	Cycles with maximum instruction issue
PAPILSTL_CCY	Cycles with no instructions completed
PAPILFUL_CCY	Cycles with maximum instructions completed
PAPILBR_UCN	Unconditional branch instructions
PAPILBR_CN	Conditional branch instructions
PAPILBR_TKN	Conditional branch instructions taken
PAPILBR_NTK	Conditional branch instructions not taken
PAPILBR_MSP	Conditional branch instructions mispredicted
PAPILBR_PRC	Conditional branch instructions correctly predicted
PAPILTOT_INS	Instructions completed
PAPILD_INS	Load instructions
PAPILSR_INS	Store instructions
PAPILBR_INS	Branch instructions
PAPILRES_STL	Cycles stalled on any resource
PAPILTOT_CYC	Total cycles
PAPILLST_INS	Load/store instructions completed
PAPILL2.DCA	Level 2 data cache accesses
PAPILL3.DCA	Level 3 data cache accesses
PAPILL2.DCR	Level 3 data cache reads
PAPILL2.DCW	Level 2 data cache writes
PAPILL3.DCW	Level 3 data cache writes

5. Countermeasures

PAPILL2_ICH	Level 2 instruction cache hits
PAPILL2_ICA	Level 2 instruction cache accesses
PAPILL3_ICA	Level 3 instruction cache accesses
PAPILL2_ICR	Level 2 instruction cache reads
PAPILL3_ICR	Level 3 instruction cache reads
PAPILL2_TCA	Level 2 total cache accesses
PAPILL3_TCA	Level 3 total cache accesses
PAPILL2_TCR	Level 2 total cache reads
PAPILL3_TCR	Level 3 total cache reads
PAPILL2_TCW	Level 2 total cache writes
PAPILL3_TCW	Level 3 total cache writes
PAPI_REF_CYC	Reference clock cycles

Table 5.4: Overview of most relevant hardware performance counters in the presence and absence of attacks, over 1 million calls to RSA and AES, as well as their rankings according to the **InfoGain** and **relief** metrics. Level 3 cache misses, PAPILL3_TCM, clearly have the strongest information for cache attacks.

Counter	AES Normal	AES w/ Attack				RSA Normal	RSA Attack		Joint Evaluation Algorithms	
	μ_n	μ_{a1}	$\mu_{a1} - \mu_n$	μ_{a2}	$\mu_{a2} - \mu_n$	μ_n	μ_a	$\mu_a - \mu_n$	infoGain	Relief
1	0.0002	0.92	0.9189	3.56	3.5598	1.12	2601.4	2600.28	0.885	0.245
2	612.33	828.60	216.27	1151.71	539.38	8.840e+07	8.956e+07	1.151e+06	0.714	0.014
3	61.93	71.01	9.08	79.93	18	2.453e+06	2.484e+06	3.1e+04	0.683	0.005
4	21.95	28.77	6.82	30.35	8.4	727.87	3417.5	2689.63	0.531	0.034
5	21.99	28.83	6.86	30.45	8.46	727.88	3417.6	2689.72	0.530	0.033
6	24.65	24.73	0.08	25.90	1.25	490.47	3253.6	2763.13	0.528	0.029
7	28.31	28.51	0.2	28.42	0.09	559.27	3325.6	2766.33	0.513	0.028
8	15.51	9.16	-6.35	11.86	-3.65	381.12	3149.2	2768.08	0.510	0.056

Legend for the performance counters: 1-PAPILL3_TCM (Level 3 cache misses), 2-Cycles (rdtsc), 3-PAPI_REF_CYC (Reference clock cycles), 4-PAPI_CA_SNP (Requests for a snoop), 5-PAPI_CA_INV (Requests for cache line invalidation), 6-PAPILL3_TCR (Reference clock cycles), 7-PAPILL2_TCM (Level 2 cache misses), 8-PAPILL2_ICM (Level 2 instruction cache misses)

The experiments are performed on an Intel Core i7-4790 CPU 3.60 GHz machine with 8 MB of L3 cache and 8 GB of RAM, with Centos 7 OS. For each counter data includes samples for 1 million encryption or decryption operations. One noteworthy observation is that, in the case of AES, the values of the counters do not seem to depend on the key. However, for the analyzed vulnerable RSA implementation, some of the parameters depend on the value of the key. This behavior can be noticed, for example, not only in the decryption time but in the number of instructions executed. In fact, the number of operations performed depends on the distribution of zeros and ones in the key. However, while the

5.2. Detection of cache attacks using hardware performance counters

number of instructions executed is not affected by the attacks, the decryption times are, as they include the extra times for cache misses.

Up to 5 or 6 counters can be selected, which are representative of the attacks, as this is the maximum number of counters readable in parallel on the platform. The number of counters that can be read at the same time also varies depending on which counters are used and the combination of them. The WEKA tool [157] is useful to decide which counters carry more information relative to the attacks. This tool was designed with the aim of allowing researchers to easily access to state-of-the-art techniques in machine learning. WEKA implements several algorithms to perform attribute selection. As inputs to the tool, only a subset among all the samples is selected (otherwise the time it takes to perform the selection increases exponentially). A random selection of 50000 instances of each of the groups, that is for AES attack and non-attack and for RSA attack and non-attack, is enough for this study. Therefore, there is a total amount of 200000 samples with information about 30 counters, each labeled with '1' for attacks and '0' for non-attacks.

The first approach is to use the `infoGain` function, which evaluates the worth of an attribute by measuring the information gain with respect to the class according with “`InfoGain(Class,Attribute) = H(Class) - H(Class — Attribute)`”, where H is the entropy. Note that the experiments are balanced between attack and non-attack “classes”, that is $H(\text{Class}) = 1$. Thus, an ideal attribute would gain 1 bit. Values around 0.5 may indicate the attribute carries meaningful information, but only for one of the algorithms or one of the attacks. Thus, L3 cache misses are not only the most meaningful predictions, but also work across the considered scenarios.

A second approach is to evaluate the relief algorithm [158] for feature selection. Unlike the InfoGain, which only evaluates information gained from each attribute individually, the relief algorithm outputs a score of the predictive value of an attribute relative to other attributes. More positive weights indicate more predictability for this attribute. To calculate the weight of an attribute, it iteratively first identifies the nearest neighbors from the same and different classes. Then, weight increases if a change in the attribute leads to a change in the class and decreases when a change in the attribute value has no effect on the class.

Finally, Table 5.4 presents a summary of the counters which give most relevant information for detection according to the selection algorithms, altogether with their mean values for the considered scenarios, and with the differences between the respective means of attacks and the *expected* behavior highlighted in gray. Both tests indicate that L3 cache misses are the most meaningful indicator. In fact, the relief algorithm scores all other attributes with very low scores, implying only little additional gain from using them.

Figure 5.12: Mean number of LLC misses over time for AES and RSA executions in the presence and absence of cache attacks. The numbers next to *flush* indicate the number of lines flushed at a time. After the start up peaks, the misses go to zero in the absence of a cache attack, while under attack they remain high.

5.2.3.2 Concurrent Signal Assessment

Tracking hardware performance events for each cryptographic operation showed that victim-based attack detection is feasible and helped identifying relevant counters. However, achieving fast detection with this approach would require adding instructions in the middle of the protected code. Hence, it requires alteration of the target code, which adds unnecessary burden on the user and diminishes practicality. Also, for more effective attack detection, it is preferable to read performance counters concurrently to the execution of the sensitive process. This way, even attacks that succeed during the execution of a single call to the sensitive function, e.g. the attacks presented in [159, 14], can be detected and prevented in time.

During the initial experiments, all implementations feature a *start up* behavior, where data is loaded into the cache for the first time and the frequency of the CPU might be adjusted. The subsequent executions feature a more constant behavior. Regarding the counters analyzed, for AES these start up executions show indistinguishable behavior under attack and without an attack. For RSA, they can be distinguished, but it would be necessary to know exactly if the current sample belongs to the start-up group of *normal* executions.

However, switching to continuous monitoring, the differences between algorithms disappear and the start-up behavior is restricted to a short time at the beginning of the execution of the processes. Figure 5.12 represents the mean value of the L3 miss counter in our continue monitoring scenario setup, for FLUSH+RELOAD attacks as well as *normal* execution of the mentioned encryption processes. The average is computed over 1000 encryptions and counters

5.2. Detection of cache attacks using hardware performance counters

are read every 100 μs . It can be clearly observed that after the initial transient state, the number of misses goes to zero in the absence of attacks (`aes no attack` and `rsa no attack`) for both crypto primitives. It can also be observed that the mean number of misses in the case of an attack varies with the number of lines flushed each time `aes 1 flush`, `aes 2 flush`.... Thus, with concurrent monitoring, both algorithms behave similarly for the *normal* executions.

Switching to continuous monitoring of the counters implies that the information on total encryption times or reference cycles is no longer useful nor available. A new analysis considering each sample collected at a period of 100 μs as an independent input to the selection attribute algorithms, using the same number of samples, ensures that the information of the other counters mentioned in Table 5.4 is still optimal for attack detection. The results show that, for the LLC misses counter, the `infoGain` increases up to 0.92, while values for the other counters decrease. Additionally, the relief algorithm output still gives better score for the L3 cache misses (0.18) and in this scenario, this value is still 5 times bigger than the weight of the next counter, indicating the `L3_TCM` is still the one counter of choice for cache attack detection.

An additional experiment determines how well a cluster algorithm would distinguish between attacks and non-attacks with the periodically sampled data from several counters at once. WEKA also includes clustering algorithms. This work includes tests with *Expectation Maximization (EM)* and *Self Organizing Maps (SOM)*, setting the number of clusters to two. The most interesting outcome of these experiments is that, while these algorithms were able to classify in the same cluster respectively 84% and 91% of the *attack* samples, when using only the LLC misses counter, this number decreases to around 50-60% when adding other counters.

These results indicate that cache attacks can be detected, regardless of the algorithm the victim process runs, by only using information gathered from the L3 cache miss counter. The studied algorithms feature zero misses after the initial warm-up, except if an attacker is forcing misses. Additionally, as all known cache attacks, including FLUSH+FLUSH cause cache misses on the victim process to obtain information, the results obtained here for the FLUSH+RELOAD attack are applicable to other attacks. Thus, this is the only attribute used, as it provides most information and keeps the detection tool simple, using as few resources as possible.

5.2.4 CacheShield

So far, techniques proposed to detect cache attacks imply monitoring the victim VM, the attacker VM, and any other VM running in the same host [67, 66].

5. Countermeasures

Previous works require monitoring all VMs at rates which vary from 1 μ s (to detect an encryption going on) to 5 ms (to reduce overhead) result in huge overheads, and increases with each new virtual machine allocated in the same host. Plus, it requires the modification of the hypervisor.

As a difference with previous approaches, the goal is to avoid monitoring all the other processes or VMs running in the same host; i.e., to focus only on the victim candidate process assuming access to the performance counters within the VMs. Although most cloud providers currently do not allow full access to the performance counters, at this time they have added functionality to provide secure and transparent access to HPCs [127, 160]. On the other hand, hypervisor systems such as VMware and KVM which support this access, can be easily configured to permit reading the counters inside the VM. Moreover, it is possible to decide which of the VMs allocated in a host would have access to the counters for their processes upon request. Even when this approach can be implemented at the hypervisor, cloud providers would prefer to enable the counters for the VMs that require it, leaving the responsibility on them, than to take care of these attacks.

By leaving the choice of deciding which processes should be monitored and when in the hands of the user, the impact in performance of such monitoring is reduced to a minimum, as the monitor only watches a possible victim when it is executing the protected task. From the cloud provider's point of view, this way of facing detection also means no waste, as it only affects the implied VM, so the owner of the VM assumes the performance penalty, and only when it is necessary. Additionally, as the user decides when it is needful to protect a process, there is no need to detect when a sensitive process is executed. As a consequence, the risk of not detecting the execution of this sensitive process is removed and the probability of missing an attack is reduced.

Figure 5.13: Overview of CacheShield.

Figure 5.13 presents a diagram of the proposed solution. Whenever a user wants protection, she informs the CacheShield module, which utilizes the information gathered from the performance counters to decide whether the user is being attacked or not. If CacheShield detects an attack, an appropriate response

5.2. Detection of cache attacks using hardware performance counters

mechanism to prevent the information leakage is put in place. Although this proposal mainly focus on the detection phase, Section 5.2.6 discusses some of the countermeasures that can be implemented to effectively prevent the attack from retrieving information, such as the utilization of a fake key or the addition of noise patterns in the cache.

The goal is to propose a detection algorithm applicable to any victim algorithm and any cache attack, not used during the training stage of the algorithm. Supervised learning algorithms, such as neural networks, have already been used to detect certain cache attacks. A supervised algorithm has to be “trained” to detect the attack. That is, it requires a labeled data set including data from the different attacks we want to detect, so they can build models of them and identify their characteristic features. The drawback of supervised learning is precisely the need to train the algorithm for each situation, for each algorithm and each attack. As a consequence, new attacks, or attacks with different patterns, would not be detected.

The alternative is using unsupervised techniques. An unsupervised algorithm does not receive labeled data, by itself tries to cluster the received data into different groups or to find relationships between different inputs in order to put any new sample in the appropriate cluster. The previous section briefly explores clustering techniques to select the counters which can identify an attack. Other kind of unsupervised techniques are anomaly-based detection algorithms, which in theory could detect “zero-day” attacks. No successful cache detection fully based on anomaly-detection techniques has been demonstrated when this proposal was presented.

Change Point Detection (CPD) methods are designed to deal with the problem of detecting abrupt changes in distributions. Under the assumption that cache attacks have an effect in the performance of the protected algorithms, change-point detection algorithms stand as great candidates to detect LLC attacks. An algorithm based on change point detection techniques, which is self-learning so it adapts itself to detect different attack patterns, allows to fix the expected attack detection delay. The algorithm is computationally simple, so it respects the constraint of minimum impact in performance, and can be implemented online.

Based on the previous results that indicate that the LLC misses counters can be enough for detection, CacheShield monitors the counters for LLC misses and for total cycles. The former gives information about the use of the LLC of the protected processes while the latter gives information about when it is running or when it has finished. According to Figure 5.13, the CacheShield module needs the Process Identifier (PID) of the process to protect and the protected process also needs to know the PID of the CacheShield process. The reason is that both processes need to communicate with each other (one needs to inform the other

5. Countermeasures

when to watch and other needs to inform the one when there is an attack going on), and that the counters can be attached to gather the data from a single process given its PID.

On Unix systems, the easiest way to use CacheShield is to use the fork operation, and then to use the *exec* system call to run the module and to give it the PID of the parent process. The parent process then can execute the desired operation while being monitored. In case that the parent process stops or waits for something, CacheShield automatically stops after noticing the parent has not been running for a while. This means that, when the parent runs again, it needs to send a “SIGCONT” signal to the CacheShield tool. In a similar way, if the tool detects an attack, it can send a signal to inform the parent. On Windows Systems the mechanisms for inter-process communications are slightly different, but the tool can be also adapted.

Change Point Detection:

In order to effectively assess the detection task, the change point detection theory (CPD) [161] is used. This theory can be used to construct the so called quick detection algorithms, which have been successfully applied for quality control, signal-processing, anomaly or intrusion detection tasks among other problems [162, 163, 164, 165]. The assumption in these scenarios is that the parameters describing the monitored system do not change or change very slowly under normal conditions. The parameters can, however, change at unknown time instants (including at startup) into anomalous conditions. Thus, CPD algorithms are used to determine if there has been a significant change in the characteristic parameters of the monitored system, quickly and with high confidence.

The theory of change point detection leads to the development of efficient algorithms presenting certain optimality properties, in the sense that for a given False Alarm Rate (FAR) they minimize the average time it takes to detect the change in the descriptive system’s features [165]. CPD algorithms can be easily implemented, do not require too much memory and, as a consequence do not have significant computation overhead. These methods belong to the “anomaly detection” class and are *unsupervised* techniques. Hence, they are well-suited to detect new attacks. All these properties make them very attractive for the attack-detection objective. The parameters of the algorithm, and how the key issues are addressed, including the choice of models or the use of prior information, are discussed below.

Denote the sequence of observations of the N random variables monitored in parallel as $X(t) = (X_1(t), \dots, X_N(t)), t \geq 1$. Before a change occurs, the joint probability distribution of the random variables $X_1, \dots, X_N$ also known as *prechange distribution*, can be denoted as $p_0(X_1, \dots, X_N)$. If a change occurs at

5.2. Detection of cache attacks using hardware performance counters

an unknown time instant λ , the observations will follow a different distribution $p_1(X_1, \dots, X_N)$, also called *postchange distribution*. That is, when $t < \lambda$ the observations $X(t)$ will have conditional pdf $p_0(X(t)|X(1), \dots, X(t-1))$, and pdf $p_1(X(t)|X(1), \dots, X(t-1))$ for $t \geq \lambda$.

Under the hypothesis that a change has occurred, the stopping time τ at which the alarm is triggered gives a measurement of the detection time. It is typically defined as the first time the change sensitive statistic watching the system, exceeds a threshold. Naming E_0 and E_λ the expectations for the sequence of observations prior and after the change at time λ , the Average Detection Delay (ADD) is defined as:

$$ADD_\lambda(\tau) = E_\lambda(\tau - \lambda | \tau \geq \lambda)$$

On the other hand, considering that there has not occurred any alarm, the mean time between false alarms will be given by the expression $E_0\tau$. As a consequence of this definition, the average frequency of false alarms or FAR is defined as:

$$FAR(\tau) = \frac{1}{E_0\tau}$$

For a good detection procedure it is expected low FAR and small values of the expected detection delay. The design of CPD algorithms often involves a trade-off between these two parameters. Page's Cumulative Sum (CUSUM) detection algorithm [153] is one of the most popular CPD algorithms: with a full-knowledge of the pre-change and post-change distributions it provides an optimal scheme minimizing the worst-case detection delay. Page's CUSUM algorithm utilizes the Log-Likelihood Ratio (LLR) to check the hypothesis that a change occurred, LLR is defined as:

$$s(t) = \ln \frac{p_1(X(t)|X(1), \dots, X(t-1))}{p_0(X(t)|X(1), \dots, X(t-1))}$$

The key property of this ratio is that a change in the parameter under study will also cause a change in the sign of the log-likelihood ratio. In other words, $s(t)$ shows a negative drift before change and a positive drift after change. The relevant information for the detection task lies then in the difference between the value of s and a minimum value. The decision rule is based on a comparison with a threshold h :

$$g_k = S_k - m_k \geq h$$

5. Countermeasures

where

$$S_k = \sum_{t=1}^k s(t) \quad m_k = \min_{1 \leq j \leq k} S_j$$

This decision rule can be replaced by the following, which obeys the recursion and whose value for the initial observation is $k = 0$.

$$g_k = \max \left\{ 0, g_{k-1} + \ln \frac{p_1(X(k))}{p_0(X(k))} \right\} \geq h$$

Then the detection time for the given threshold is

$$\tau(h) = \min \{k \geq 1 : g_k \geq h\}$$

Although this first approach considers that both distributions are known, this assumption is usually not true, and as a consequence this proposal has to be adapted for each situation. One distribution in advance or none may be known, so it may be necessary to estimate the parameters of the algorithm during the runtime. As long as the estimators for the distributions and the real observation meet certain convergence conditions, it will be fixable; for example, the desired detection delay or the FAR.

Change Point Detection in CacheShield:

While facing the cache attack detection, the attack may start from the very beginning or it may start after a few “normal” transactions. Both situations are efficiently managed by the proposed CUSUM algorithm. Let assume that each new sample can be classified into one of two different groups or clusters, namely “attack” and “non-attack”. The “non-attack” cluster represents the expected behavior of the protected process under normal conditions. Based on the information from the performance counters, this assumption is that, after a few samples corresponding to the initialization of the protected process, the number of L3 cache misses will be around 0, then $\mu_{na} = 0$. On the other hand, when there is an attack, the mean number of misses is μ_a . Then, each new sample belonging to the “attack” cluster will be around μ_a . The value of μ_a is unknown and depends on the attack, so it needs to be computed and recalculated with each new sample.

Let denote as $miss_i$ each new sample that the CacheShield module gets referring to the protected process. The algorithm decides if it belongs to one cluster or to the other. To do so, the algorithm calculates the value of the “probability” that $miss_i$ belongs to each one making use of the distance metric. This way the distance from $miss_i$ to μ_{na} is defined as:

5.2. Detection of cache attacks using hardware performance counters

$$d_{na}(i) = miss_i - \mu_{na} = miss_i$$

Then, the distance with the “attack” cluster is

$$d_a(i) = |miss_i - \mu_a|$$

As stated before, the value of μ_a is unknown when starting to monitor the process. An arbitrary initial value is set. Whenever a new sample $miss_i$ is obtained, if $miss_i \geq 0$, the algorithm updates the value of μ_a as follows:

$$\mu_a = (1 - \beta) * \mu_a + \beta * miss_i$$

This method is known as exponentially weighted moving average, where the weight of the older datum decreases exponentially. This way of estimating the mean of the “attack cluster” makes the election of the initial arbitrary value irrelevant after collecting a few misses samples. If the initial value is chosen too low, it may trigger false positives. The election of an initial value should be higher than 10, in order to keep the rate of false positives low, while being able to detect the attack in a reasonable time. The next section discusses the noise tolerance of the proposed detection algorithm. The experiments use a $\beta = 0.05$ and an initial value of 12.5.

Therefore, the probability of belonging to each cluster is defined as:

$$p_{na}(miss_i) = \frac{d_a(i) + 1}{|d_{na}(i)| + |d_a(i)|}, \quad p_a(miss_i) = \frac{d_{na}(i) + 1}{|d_{na}(i)| + |d_a(i)|}$$

The value 1 has been added to avoid divisions by 0 in the LLR calculation that has to be performed as part of the detection algorithm. As a result, for every sample k , $k \leq 1$ the detection rule is expressed as follows:

$$g_k = \max \left\{ 0, g_{k-1} + \log \frac{d_{na}(k) + 1}{d_a(k) + 1} \right\} \geq h$$

As it can be easily derived from the previous equation and according to the properties of the LLR, when the number of misses is 0 or close to 0, the distance between the sample and the “non-attack” cluster $d_{na}(k)$ will be lower than the distance to the attack cluster $d_a(k)$, so the value of the metric g_k decreases or stays at zero. On the other hand, readings from the LLC misses counter approaching to the attack cluster increases the value g_k . The properties of this approach enable the threshold selection based on a minimum detection time. Note that, when the error in the estimation of the mean ϵ approaches to zero, $d_a(i) = \epsilon$ also tends to zero, then the increase in the value of g_k is also limited

5. Countermeasures

$$\log \frac{d_{na}(k) + 1}{d_a(k) + 1} \leq \log(\mu_a + 1)$$

As a consequence, the minimum expected detection time for the given threshold h is:

$$\tau_e(h) \geq \frac{h}{\log(\mu_a + 1)}$$

or reformulating this equation, we can get the threshold h , for a minimum expected delay τ_e

$$h_\tau \leq \tau_e * \log(\mu_a + 1)$$

Algorithm 6 CacheShield detection algorithm

Input: Process PID

Output: *Attack detected*

```

read_counters(misses,cpu_cycles);
wait;
while victim_is_running do
    read_counters(misses,cpu_cycles);
    if misses > 0 then
        update  $\mu_a$ ;
        update  $h$ ;
    end if
    calculate  $g_k$ 
    if  $g_k > h$  then
        trigger_alarm;
    end if
    wait;
return detected;
```

The unit of the $\tau_e(h)$ is *number of samples*. Given that the most effective cache attacks can potentially extract most of the key with just one execution of the victim, the sampling rate must be chosen lower than the execution time of the victim. As the execution time of these algorithms is in the order of few milliseconds, a sampling rate of 100 μs seems sufficient to provide evidence of the attack. This frequency can be increased at additional load for the system. For example, for an expected detection delay of 1 ms with a sampling rate of 100 μs we can define the threshold as $h = 10 * \log(\mu_a + 1)$. As a result of this selection of h , when the μ_a is recalculated, the threshold should be recalculated too. The choice of the threshold h also determines the tolerance to noisy frames, and as a consequence the false positive rate. In practice, the false positive rate cannot be estimated and has to be measured.

5.2. Detection of cache attacks using hardware performance counters

Figure 5.14: Relevant parameters for the detection task, the values correspond to an execution of PRIME+PROBE attack on AES

Algorithm 6 summarizes CacheShield implementation and an example of the values of the parameters considered in the detection process is given in figure 5.14. This figure represents a real attack (PRIME+PROBE) trace against AES. As it can be observed the value of the estimated μ decreases until is almost similar to the real misses trace. At some point, when the measured trace is more similar to the μ_a than to the non-attack trace, the value of g_k increases until it eventually reaches the threshold around the 80th sample.

5.2.5 Evaluation of CacheShield

In this section we evaluate the performance of CacheShield. To this end we ran several experiments in different environments and machines.

Native Environment The experiments for non-virtualized environments were performed in an Intel Core i7-4790 CPU 3.60GHz machine with 4 physical cores (8 virtual), 8 MB of L3 cache and 8 GB of RAM, with Centos 7 OS.

KVM-based Hypervisor These experiments used the same hardware as above, but this time within a VM also with Centos 7 hosted in KVM as hypervisor.

VMware-based Cloud Server We have also executed experiments in a host managed with VMware, this machine is equipped with a Intel XeonE5-2670 v2 processor (2.50GHz), with 10 physical cores (20 virtusl), 25MB of L3 cache and 32GB of RAM. The OS in these VMs was Ubuntu 12.04.

When a user is executing the crypto algorithm in their own personal machine, he can configure the performance counters to get information about the utilization of the whole machine or about particular processes running at that time. However, when executing the crypto algorithms in cloud environments they can not get any information about what their neighbors are doing. In such scenarios, it becomes

5. Countermeasures

mandatory to study how the execution of different applications running in parallel with the protected process affects the behavior of CacheShield. Note that we only use the "total cycles" counter to determine if the victim is executing or not and the LLC misses counter is the only one used to decide if there is an attack going on. Consequently, applications consuming high amount of memory resources are likely to cause the LLC misses indicator to rise, and as a consequence, to trigger false positives. For this reason, we have selected several worst case scenario applications with high memory activity to run in parallel with the *victim* and CacheShield:

Yahoo Cloud Serving Benchmark This benchmark was originally designed as a tool that provides a common evaluation framework and a set of common workloads to test the performance of different serving stores as elastic search, Cassandra, MongoDB among others [145]. It allows different configurations for the workloads and provides a set of *example* workload scenarios, together with a workload generator, which generates the load to test storage systems. In our experiments, we used this benchmark with the Apache Cassandra database and the example workload named *workloada*.

Video Streaming Another kind of application that can generate cache misses is web-browsing or video streaming. The video streaming VM continuously streams and plays back youtube videos on the firefox browser.

Randmem Benchmark This benchmark was originally intended to test the impact of burst reading and writings [166]. Depending on the configuration, the benchmark access data stored in an array either sequentially or in random order. The tool also allows to configure the size of the memory it is going to use, by default it tries to use as much as possible, up to 2 GB. In our experiments, we launch each **randmem** instance with no memory limitation, which means 2 GB of RAM memory are used by each instance. We configure Randmem to perform both reads and writes with a random order.

To show the applicability of CacheShield to a broad range of implementations that require protection against cache attacks, we chose from a range of crypto primitives and implementations, though focusing on vulnerable ones, since such legacy implementations actually require protection. The three crypto algorithms considered as *victims* are:

AES: we consider the vulnerable T-Table implementation of AES from Openssl 1.0.1f, which is fast, but also leaky.

RSA: we analyzed the RSA implementation from Openssl 1.0.1f, with a 2048 bit key, and the `RSA_FLAG_NO_CONSTTIME` flag set.

5.2. Detection of cache attacks using hardware performance counters

ElGamal: we chose the ElGamal [167] implementation of libcrypto 1.5.0 with a 4096 bits key. ElGamal is a public key cryptosystem that is based on the Diffie–Hellman key exchange. During the decryption phase, it performs a modular exponentiation among other operations. This particular version of libcrypto uses the square-and-multiply algorithm. Unlike AES and RSA, ElGamal was not considered during the design of CacheShield, and hence shows how CacheShield can be expected to perform for other types of algorithms.

These algorithms differ quite significantly in their particular implementation and usage of cache. Many other potentially leaky codes might require protection, and we are confident that CacheShield will perform well.

The design of the tool was actually decided using samples of one of the possible attacks (FLUSH+RELOAD). However, we tested different algorithms. We have implemented and executed three popular attacks, namely FLUSH+RELOAD, FLUSH+FLUSH and PRIME+PROBE, in order to evaluate the effectiveness of CacheShield across different types of cache attacks. These attacks targeted the above-mentioned algorithms, for each of them we collected data under attack as well as from normal executions, as baseline behavior. Under each configuration, we have collected data for more than 1000 executions of the crypto primitives, and, in the case of the AES attack, we have also considered different attack rates (number of lines flushed at a time). The attacker may try to gain different amount of information from the T-tables per execution [64].

Note that the attack requirements for FLUSH+FLUSH, FLUSH+RELOAD and PRIME+PROBE differ significantly. While FLUSH+X attacks are faster and more precise, they require shared data, i.e. deduplication between attacker and victim. All the attacks performed in virtualized scenarios were across VMs so we enabled deduplication features (KSM and Transparent Page Sharing (TPS) for KVM and vmware respectively) to perform FLUSH+X attacks. PRIME+PROBE attacks work across VMs even without deduplication, so we disabled deduplication and enabled huge pages.

PRIME+PROBE attacks require, prior to the information extraction, a *profiling* of the cache [71, 79, 80]. The profiling stage reveals the sets the victim process is accessing and that carry the necessary information to succeed in the attack. In this situation, the detection tool will trigger an alarm whenever the set being tested by the attacker was actually used by the victim. Figure 5.15 visualizes the output of the detection algorithm, for the cache profiling stage of an 8 MB L3 cache when the target is the T-table implementation of AES. The x-axis represents each set of the cache being evicted during the PRIME+PROBE profiling step; a 1 on the y-axis indicates that an alarm has been triggered. Thus, alarms are only triggered when the cache attack affects the target. For instance, if we observe in the figure 5.15 around the set 4000 there is a wider

5. Countermeasures

Figure 5.15: Output of CacheShield when the cache is profiled accessing each set. "1" indicates a positive attack detection.

peak. This region is where the T-Tables are actually located in the cache.

As stated in previous sections, the main characteristics defining the detection algorithm are the mean detection time, and the false positive rate. Table 5.5 presents the results for mean detection time under different configurations, for the different attacks and algorithms and table 5.6 shows the results related to false positives in noisy environments. Both table highlight the results for the ElGamal encryption. For all evaluated attacks, the detection rates are 100%.

Table 5.5: Mean detection time (ms) per attack and scenario for the evaluated crypto algorithms. Note that in all cases CacheShield has the same configuration and that detection times are much lower than the ones required for the attack to succeed. The mean was computed with at least 1000 samples per scenario

Scenario	AES			
	F+R (1)	F+R (4)	F+F (1)	P+P
Native	3.98	4.48	3.38	5.08
KVM	7.38	7.05	6.64	9.53
Vmware	8.75	5.98	10.74	13.42

Scenario	RSA			Scenario	ElGamal		
	F+R	F+F	P+P		F+R	F+F	P+P
Native	3.70	3.54	5.16	Native	2.97	3.47	3.68
KVM	4.08	3.92	4.93	KVM	3.76	3.45	3.98
Vmware	4.43	3.87	4.51	Vmware	4.83	5.06	7.08

Note that the sampling rate is $100 \mu\text{s}$ and that we want to detect the attack before the end of each decryption (for public key cryptography). If we wished to detect attacks against algorithms whose duration is below 5 or 6 ms, we would need to increase the sampling rate, since mean number of samples required to detect the attack cannot be lowered arbitrarily without increasing the FAR too much. The duration of the decryption/encryption depends on frequency of the processor, and as a consequence on the machine. For example, ElGamal encryp-

5.2. Detection of cache attacks using hardware performance counters

Table 5.6: False positive rate for different scenarios and algorithms. (Instances: **Y** - Yahoo Cloud Serving, **V** - Video Streaming; **R** - Randmem)

Scenario	Noise Instances			False Positives		
	Y	V	R	AES	RSA	ElGamal
KVM	1	0	0	1.2%	4.5%	2.8%
KVM	0	1	0	1.1%	3.4%	0.6%
KVM	0	0	1	12.2%	21.4%	15.4%
Native	1	0	0	1.2%	4.1%	2.4%
Native	0	1	0	0.5%	1.3%	0.3%
Native	0	0	1	11.1%	19.2%	13.8%
VMware	1	2	10	0.1%	5.9%	4.1%

tion takes around 11 ms when being attacked on the i7 machine, while this time increases up to 24 ms on the Xeon machine. Thus, we are able to detect attacks against ElGamal when less than the 37% of the encryption has been performed for the i7 machine, and 30% for the second machine in the worst case. Regarding to RSA this mean execution times are around 18 ms for the i7 machine and around 37 ms for the other. Then, in the worst case, on average we detect the attacks with less than 50% of the decryption performed in the first case and with about 37% of decryption in the second one.

Regarding to the existing differences between false positive rates for AES and public crypto algorithms, these are easy to explain. While between AES encryptions exists some time in which the processor does nothing, the others execute uninterruptedly. This fact increases the probability of other processes accessing the cache during the same interval. To give one example, while the AES encryption in the period of 100 μ s is only active during around 7000 cycles while the RSA process is active during about 30000 cycles for the VMware machine when there is no attack. Thus, the collected noisy traces look differently. Figure 5.16 depicts the LLC misses for one noisy RSA encryption, besides the initialization steps, it can be observed a high amount of cache misses during the whole decryption. Similarly, Figure 5.17 corresponds to one process performing AES encryptions. In this second case, misses are observed punctually.

The results also show that the tolerance to noise of the detection algorithm is more dependent on the hardware than on the virtualizing technology: While the results for the native and KVM scenarios are similar and the hardware is the same, the results are significantly better on the Xeon machine. The Xeon machine did not trigger any false positives when there were one or two VMs generating "noise" concurrently, until we launched several more instances. As this machine is more similar to the kind of machine cloud providers utilize, these results show that the tool is practical in these environments.

5. Countermeasures

Figure 5.16: LLC misses for a complete RSA execution under randmem benchmark. Although there is no attack going on, the randmem benchmark causes multiple misses. In this case, CacheShield does not distinguish whether they are due to an attack or not.

Figure 5.17: Sample of a normal execution of AES under randmem benchmark. The process is monitored during the time an attack would have taken. Note the differences between the AES and the RSA processes.

One approach to reduce the false positives in noisy environments could be considering the variance of the samples collected in the CUSUM algorithms proposed, as attacks present low variance compared with noise. However, we could fail to detect attacks masquerading as memory activity by generating different number of misses each time. Another consideration relative to memory utilization, and the false positives that are triggered when memory utilization is high is that PRIME+PROBE attacks need low memory activity to accurately locate the sets and to perform the attack, other way it renders much more difficult. On the other hand, FLUSH+FLUSH attacks are more tolerant to noise, but memory activity degrade its performance. So it is not likely that the attacker performs the attack in a situation where the memory is highly utilized. Additionally, the level of utilization of public clouds is low [146], so the assumption of high memory utilization in the considered cloud scenarios may not be realistic. We only studied it as the worst case scenario. As for using the tool in our controlled physical machine, we can get to know which is the level of utilization of the memory and then, decide if it is worth it to change the parameters of the detection algorithm.

One last consideration about our tool is the amount of CPU it utilizes to monitor the victim and compute the detection algorithm. We show that, as desired, the tool consumes negligible amount of resources. Figure 5.18 and figure 5.19 show the mean CPU utilization of CacheShield for different sampling rates

5.2. Detection of cache attacks using hardware performance counters

Figure 5.18: CPU utilization of the i7 machine in native environment for different sampling rates in microseconds. Highlighted the 100 μ s rate as it is the one we use in our experiments

Figure 5.19: CPU utilization of the Xeon machine in virtualized environment for different sampling rates in microseconds. Highlighted the 100 μ s rate as it is the one we use in our experiments

and for different situations, namely when the victim is attacked and when it is not, because the amount of operations it has to do changes, and again for both architectures, depending on the sampling rate. To obtain the CPU utilization we have measured the time it takes to read the counters and perform the calculations and the total time elapsed. The utilization is given by its division. Note that sampling rates of 10 μ s are not always achievable as sometimes (around 10% of the time) it takes more time to read the counters and perform the calculations. For our particular configuration of the tool, in both cases total utilization of CacheShield is below 5% of CPU utilization (of one logical core).

5.2.6 Cache Attack Countermeasures

Once an attack has been detected, CacheShield needs to react in some way. One way is to simply interrupt the monitored process and to purge used keys. While this approach ensures high security, it decreases the usability, as any false positives will result in a total cryptosystem shutdown. An alternative is to continue execution, but to apply preventative measures to reduce or prevent the exploitability of the cache.

5. Countermeasures

Adding Noise A simple method to hinder cache attacks is making the channel noisier, e.g. through frequently flushing cache lines used by the protected process, or by performing additional reads on data. This approach works particularly well if critical data is known, e.g. the tables of an AES implementation. A similar approach (currently included in some distributions) includes pre-loading the sensitive data into the cache.

Dummy Operations An alternative approach is to perform dummy operations on meaningless secrets. In practice this can mean to run the protected process with a newly generated secret. The original process can either be paused, or be continued in parallel to the dummy process. Parallel processing obfuscates the true leakage. However, depending on the attack type, an attacker might still succeed with an increased number of observations. Pausing has the advantage that the attacker might actually extract the dummy key and discontinue the attack. The monitor can then restart the original process in the absence of the attack. Either way, the performance degradation is not negligible, but it only is incurred in the presence of an attack.

Protected Implementations The main reason why leakage is still observed in security solutions is the performance overhead that pure constant time implementations present. A way of avoiding such a scenario is to use protected implementations *only* when CacheShield detects an attack behavior. When no attack is detected, faster (less secure) implementations can be used.

Other more sophisticated solutions are also possible, but might not be as universally applicable. Since our focus is on the lightweight detectability of cache attacks at the user level, we do not explore these additional avenues of countermeasures.

In summary, CacheShield monitors the victim process upon when active and by users request. The analysis of various hardware performance counters revealed that the LLC miss counter by itself carries enough information to detect cache attacks. This fact allow us to implement an efficient algorithm that maximizes fast and reliable attack detection, while minimizing false positives and keeping the performance overhead minimal and restricted to the victim VM. Indeed, it succeeds in detecting all existing cache attacks, including stealthy attacks that are ignored by other solutions, since our proposal uses attack characteristics that are independent of attack and victim behavior. The tool can be utilized by any user, since it does not require hypervisor or OS support. No help from the underlying system is needed rather than access to the HPCs.

5.3 Detection of Other Microarchitectural Attacks

In section 5.2 we showed that it is possible to detect cache side-channel attacks using performance counters. While cache side-channel attacks are an immediate threat due to their ability to extract sensitive information from co-located VMs, they do not represent the only threat in shared environments. Another class of attacks are performance degradation attacks, by inducing intense resource contention attackers are able to degrade neighboring applications performance.

These attacks do not directly break confidentiality. Instead they can make a service or app annoying to use and increase the response time of an e-commerce place up to 38 times. One of the immediate consequences of these denial of service attacks is that a user may desist of the use of the service or even switch to a competitor service. Besides, in cloud environments, the performance of all the VMs allocated in the same host is effectively degraded.

We show that by using the same approach that we have used before, that is, by changing the configuration of CacheShield and adjusting some of its parameters, we can also detect these degradation attacks.

5.3.1 Memory DoS attacks

Memory DoS attacks refer to the Degradation of Service caused by contention of hardware memory resources. The ring bus that connects elements in some processor sockets (e.g Intel architectures), is hard to saturate and difficult to lock with normal operations. However, there are certain operations known as exotic atomic operations which can lock the internal memory buses. Since the processor has to ensure atomicity while processing these operations, it has to lock the memory. Thus, an attacker continuously performing these operations effectively degrades performance of real cloud applications in commercial clouds [106] and Android devices [107].

Assuming that an attacker has the ability of placing a VM with its victim, he can intentionally abuse hardware resources to degrade the performance of all the VMs running on the same host. Previous works have also considered the cache as a possible mean of degrading the performance of the victim by accessing the same sets it is using. However, as we have previously demonstrated the feasibility of our approach to detect cache attacks or abuses, we focus on exotic atomic locking attacks. We demonstrate that when performance counters access is enabled inside the VM we can measure the performance of the process we want to protect, and as a consequence, we are able to detect deviations in the expected behavior caused

by attacks, just adapting the CacheShield approach.

5.3.2 Preliminary study

We consider two different applications commonly found in cloud environments: web servers and databases. As representative applications of each group we have selected Apache web server and Apache Cassandra (NoSQL database). These applications once installed and started launch multiple threads to answer the requests they get. This means there can be multiple PIDs associated with the application, so we need to change our initial approach. We assume that our VMs are dedicated servers, which means they are only used for web serving or as a database.

Since we still want to monitor the applications concurrently with its execution without modifying them, we decided to monitor the whole VM. PAPI allows to detect the number of cores of the virtual machine and to attach a group of counters to each core similarly as it does for each PID. However, monitoring the whole VM from the inside, requires root privileges. If the hypervisor has enabled the use of performance counters, we assume that the owner of the VM has such privileges. Additionally, in this scenario detection time is not as critical as for cache attacks, so we have selected a new sampling rate of 100ms.

We gathered the information for the analysis in the VMs virtualized with KVM running in the i7 processor and the VMs virtualized with VMware running in the Xeon machine. For both applications we have considered different request rates and in the case of Cassandra, different kinds of operations (adding, removing, retrieving and modifying entries of a database). Figure 5.20 shows the distributions of the values of relevant counters gathered every 100 ms referred to the Apache web server in the i7 machine, including more than 50000 samples each. At request rate 1, the VM receives two requests each 100 ms whereas at request rate 2 it receives and processes 4 requests per 100 ms second. Both applications and machines behave similarly, in the sense that the values of the counters are different but the trend is the same. Based on these experiments, we can derive the following conclusions:

- The values of the counters depend on the number of requests the application is processing and on the particular data the application has to process. If the request rate is unknown, the values read from the counters by themselves are not indicative of an attack, while if it is known, the counters evidence an increase of cache misses, cycles stalled and total cycles employed by the monitored process and a drop in the number of instructions executed and L3 cache access (col. 1 of fig. 5.20).

Figure 5.20: Distributions of the values of different counters gathered each 100 ms for different request rates related to the Apache web server application with and without DoS attack executed in the i7 machine virtualized with KVM.

- If we consider ratios as cycles stalled per L3 cache access, cycles per instruction or even normalize respect both the sampling rate and another counter, we can distinguish between attack and non-attack regardless of the request rate. This means that getting once the model for our application is enough, avoiding the need to acquire a different model per request rate (col. 2 of fig. 5.20).

Note that the ratios considered are also a measure of the efficiency of a processor (its architecture) executing the profiled application. The number of cycles per instruction or the number of cycles per memory access are parameters that depend on the architecture, and on the application. Although for these two particular applications it is still possible to distinguish between attack and non attack regardless of the application, we need two different models for the two architectures (i7 and Xeon). For other applications, we should get one model per architecture and application. We include the figure 5.21 that refers to the Xeon processor for comparison.

5. Countermeasures

Figure 5.21: Distribution of the ratios of the values of different counters gathered each 100 ms related to the Apache web server application, and cassandra application with and without DoS attack executed in the Xeon machine virtualized with vmware

5.3.3 Detection algorithm

As mentioned before, we can not assume the normal or expected behavior of our processes, as we did for cache attacks. We need a model of the expected behavior, that is, a mean and variance of the distributions of any of the ratios seen in figure 5.20 (e.g cycles stalled per L3 access) as according to our previous study any or them can be used to detect memory DoS attacks. The distributions of the attack and non-attack samples are clearly distinguishable.

In this scenario, we select a “limit” of degradation we are able to tolerate. This limit needs to be calculated offline and given to the system. It is hard to establish a fixed limit in this case unless the application becomes unable to keep up with the number of requests it is receiving. Otherwise, such limit is selected according to the perception a user has, or the minimum desired performance. It can be defined, for example, as the mean plus certain times the variance of our model. In our case, we have selected our limit as the mean of each distribution plus two times its respective variance. Samples below this limit will be used to calculate the “non-attack” distribution while samples over it will be used to

compute the "attack" distribution. Note that if we are too restrictive with this limit, the number of false positives will increase. As before, for each new sample, we compute the probability of belonging to each cluster based on the distance from each new sample to the mean value of each distribution, and calculate the parameters of the algorithm (g_k, h, μ_a and μ_{na}).

5.3.4 Evaluation

The experiments are performed in the same machines as were the previous ones. Given that the results of the experiments performed in the native environment and in the VMs with KVM-based Hypervisor are similar (since they are run on the same underlying hardware) this time we only consider the experiments in virtualized environments.

In the same way we considered neighbor applications for cache attacks, in the evaluation of the adapted detection system we have also considered different noisy instances. We refer back to section 5.2.5 for further details about these instances. These experiments also measure how neighboring VMs or processes degrade the expected performance of a system.

Results regarding detection time and false positives are shown in tables 5.7 and 5.8. All the experiments were performed in VMs allocated in the i7 and Xeon machines. We have considered different varying request rates, executed each experiment for at least 10 seconds and executing more than 1000 simulations per scenario (application-hardware-attack/non-attack) and achieving a detection rate of 100%. Although all the counters in the Figures 5.21 and 5.20 can be simultaneously monitored, results in Table tables 5.7 refer to cycles stalled per instruction. When a second ratio is considered (for the noisy experiments), we use cycles stalled per LLC access.

Table 5.7: Mean detection time (s) per attack and scenario for the evaluated cloud applications.

Scenario	Web server	no-SQL database
KVM	1.78	1.85
Vmware	1.97	2.04

The false positive rate for DoS attacks is lower than for cache attacks. This is mainly due to two factors, one is that we are more permissive with the degradation and the other is that the fact that performance is much more degraded by the attack than it is by neighbor applications. As before, randmem is the most degrading application. However, considering the information from cycles stalled per LLC access and cycles per instruction at the same time, we achieve a false

5. Countermeasures

Table 5.8: False positive rate for different scenarios and applications. (Instances: **Y** - Yahoo Cloud Serving, **V** - Video Streaming; **R** - Randmem)

Scenario	Noise Instances			False Positives	
	Y	V	R	Web server 1/2 counters	Cassandra 1/2 counters
KVM	1	0	0	0 %	0%
KVM	0	1	0	0%	0%
KVM	0	0	1	4.3%/0%	3.1%/0%
VMware	1	2	10	1.6%/0%	2.3%/0%

positive rate of 0%. When such attack is detected, the victim should inform the provider about the degradation it is suffering and request a migration of its VM as soon as possible.

Attacks

In chapter 5 we presented some countermeasures intended to detect state of the art cache attacks. In contrast, this chapter presents our contributions to improve the existing attacks or to the development of new attack techniques. For instance, section 6.1 describes techniques that can be used to detect co-residency of VMs in the cloud. This is a necessary step before performing a cross-VM cache attack. The remaining sections describe improvements to traditional approaches and develop new attack vectors.

6.1 Inferring the CPU load of a host from a process running on it

Since its inception, cloud computing has experienced an exponential growth. Big companies such as Amazon, Microsoft or Google have become cloud computing providers, with both government and industry using public cloud platforms as clients. Cloud computing enables on demand access to computing resources, allowing users to adapt their computing capabilities to their demands and eliminating for them the need to acquire and maintain expensive IT infrastructures. In this scenario, where cloud customers pay for the resources they use, several virtual machines run in the same host. This sharing of physical resources between tenants allows cloud providers to achieve economies of scale, reducing the operations and maintenance cost of IT infrastructures.

The isolation between VMs running on the same physical machine is provided by a virtual machine manager and supervised by a hypervisor. However, as we have already described, the underlying hardware creates covert channels between VMs or processes, that can be exploited to break the logical isolation between

6. Attacks

tenants. Side-channel leakage can be used to extract private information from co-resident VMs [14, 31] such as keys from cryptographic algorithms, as well as to detect the execution of processes like *sshd* in co-resident VMs [168]. For all these side-channel attacks to succeed, there is one requisite: co-residency. Prior to the attack, a potential attacker has to determine if the target is allocated on the same host or not.

There have been several successful attempts to detect co-allocation. In 2009, the work from Ristempart et al. [9] paved the way for the upcoming works in co-residency detection and cross VM side-channel attacks. They proved that co-residency is achievable and detectable in the Amazon’s Elastic Compute Cloud (EC2). Since this moment, most of the subsequent works demonstrating cache attacks assume that co-residency is possible and take it for granted. Although the techniques they used to determine the placement of the victim have been obfuscated by Amazon and other cloud providers, new techniques are coming up [70].

Figure 6.1: Example of the expected CPU load of a system when a process is running or not. The upper figure shows with 0 the instants when it is not running and with 1 the instants when it is running

We consider the problem of co-residency detection from a different perspective. When a process or VM is running, it consumes CPU cycles, modifying the CPU utilization of the host. This suggests that there must exist a correlation between executions on the potential victim virtual machine, and changes in the CPU load of the host. This relation is depicted in Figure 6.1. An attacker could trigger the execution of the victim, and use the knowledge of the CPU load of the system to determine if they are in the same host or not. Observing the changes in the trends of the load and its correlation with the requests the attacker is sending, should be enough.

However, cloud providers do not facilitate information about host utilization to their clients. Users can only access usage statistics related to their own virtual

machines. Thus, we have to devise a technique to gain information about the host utilization. One of our main objectives of this section is the development and validation of accurate models of the current CPU utilization of a host which can be obtained within a VM running on it.

When a process shares the CPU with others, its execution time varies. As a result, we can use this variation to estimate the load of the host. Our previous work [156] and particularly the analysis of Figure 5.6 suggests that this relation may be linear, so we first employ regression techniques (*Least Absolute Shrinkage and Selection Operator (LASSO)*) to obtain the model. In a second approach, we use Genetic Algorithms (GA) to tune the inputs of the *LASSO* in order to get simpler models with similar error rates.

In summary, we show that it is possible to obtain accurate models of the CPU load within a process or VM running on it, by just executing and measuring a test process. Our models estimate the load of the host with a Mean Squared Error (MSE) below 28% when calculating the CPU load from a process running native on the physical machine, and below 38% when the calculations are performed inside a virtual machine (worst case, *LASSO* regression).

6.1.1 Previous approaches

By definition, cloud computing shares the same physical machine between multiple client VMs, increasing the utilization ratio of the machine and, as an indirect outcome, creating co-residency data leakage problems. At this point, we must remember how CPU caches, which are shared between all the CPU cores and consequently between all the VM running in a host, can be used to extract information from a neighbour VM. They have been used to infer AES [79], RSA [71], EDCSA [58] and other cryptographic keys. In this section, we discuss some of the previous approaches that have tried to detect co-residency between VMs or to use the information retrieved from the CPU.

Ristempart et al. [9] were able to determine the internal cloud infrastructure (of the Amazon EC2). They sent ping requests to different instances and used the response timing information to create a map of the infrastructure. Zhang et al. [149] used the L2 cache side-channel with defensive purposes. They built a tool able to detect co-resident foe VMs. The techniques they described can be adapted and used to detect co-resident "victim" machines.

Zhang et al. [8] demonstrated attacks between tenants on commercial Platform-as-a-Service (PaaS) clouds. They also studied the attacker's ability to co-locate an instance with a victim. The strategy they described consists of two steps: first, the adversary has to launch a large number of instances on the cloud service, and second, he attempts to determine somehow whether any of the instances

6. Attacks

co-locates with the victim instance.

Six years after the seminal work of Ristenpart et al. was published, Inci et al. [70] revisited the co-residency detection problem. They tried to replicate the same experiments described by Ristenpart et al. and found that Amazon had obfuscated that source of information. They tried some other alternatives such as creating contention with the AES-NI instructions, but they found that the LLC was the most reliable way to determine co-location. They *primed* one set in one instance and caused access in a different one. The first instance then *probed* that set and determined that the second instance is located in the same host if the *probe* times are higher than a threshold. They also observed that instances launched within short time intervals from each other are more likely to be co-located.

We suggest using CPU utilization as a mean of co-residency detection, but as a previous step, we need to prove that it is possible to estimate the current CPU load of the host machine within an instance. In this section, we focus on the process of obtaining of CPU utilization models which can potentially be used to determine co-residency. The idea of using the CPU utilization as a side-channel has also been explored before.

Masti et al. [154] found that, in modern multi-core platforms, the isolation techniques based on dedicated cores can be circumvented using thermal channels. They were able to detect processes which were executed in neighboring cores by monitoring the temperature sensors of the cores and quantifying the heat propagation as an effect of running CPU intensive applications. In a similar way, our work studies the effect that running different loads in a host has in a test process.

Additionally, Liu [169] used a well-known relation between CPU load and temperature to estimate the load of the host in a cloud environment, also by reading temperature sensors. Their objective was conducting a study of server utilization in public clouds. Thus, a model of CPU utilization, like the one we propose, can be used to estimate utilization in public clouds. Regarding temperature sensors, they have to be exposed by the hypervisor in order to be read inside the VMs, and they are not always exposed. In fact, we have tried to read them using KVM as hypervisor, with the default configuration settings, and we have not been able to. In contrast, our approach makes use of time counters, which are available inside VMs, to calculate the total load of the host.

We propose to use the LASSO regression algorithm to find a relation between the histogram of the execution times of a test process and the CPU load of the host. However the size of the bins and the number of inputs are not optimized. For this reason we use Genetic Algorithms to find optimum values of those. This approach is also common in the literature. For example, Adams et al. [170] used

Figure 6.2: Histograms of the AES encryption cycles (time) under different CPU loads. Each histogram includes 10000 samples

GA to reduce the number of input features for different modeling algorithms, including *LASSO*. They select 24 relevant questions from a questionnaire with 123 questions. Although our target is different, their work shows that our proposed optimization is feasible.

6.1.2 Our approach

In the same way that the cache is shared among all the cores, when a process is executed simultaneously with other processes in the same physical machine, they share the CPU. As a consequence, execution times of the same process vary between executions. To measure and quantify this effect, we propose to execute in a loop a reference process and measure its execution time. In low load conditions,

6. Attacks

the test process must be a voracious CPU consumer, which tries to use all the available CPU. These kind of processes will fight with others for resources if necessary.

We have selected *AES encryption* as our reference process. As observed before, it is a high CPU consumer process, it is quite fast (we can perform several encryptions in milliseconds) and, additionally, the time it takes to perform an encryption is almost constant. When monitoring, we execute encryptions one after the other and measure the time it takes to finish each of them. Then we build an histogram with the samples collected during a fixed time interval. Figure 6.2 depicts different histograms of the encryption time under different conditions of CPU load. These histograms clearly show that the performance of the encryption process is clearly degraded by other processes. We define the CPU load as the number of cycles the CPU was executing some task divided by the number of passed cycles (considering all the cores), and multiplied by 100. As a consequence, the CPU load will take values between 0% and 100%.

As we have previously observed in Figure 5.6, if in parallel we analyze Figure 6.2, it seems that the number of encryptions lasting around 350 CPU cycles decreases linearly with the increase of activity in the host machine. When there is no load in the host, around 100% of the encryptions are in this interval, while almost no encryption lasts for that time when the host CPU load is close to 100%.

Taking as an input the distribution of the times it takes the processor to perform an encryption, we try to fit a linear model to calculate the host CPU load using a linear regression techniques, *LASSO*. For this step, we select the limits of the histograms between 100 cycles and 1700 in steps of 20 cycles; that is, we have 80 inputs for the *LASSO* algorithm. This range covers the observed times in the different machines we have used for these experiments. Since 80 different values would require at least 80 multiplication operations to retrieve the CPU of the host, we design strategies that allow us to reduce this number. In order to reduce the inputs and to still be able to calculate the load of our host, we combine the *LASSO* with a Genetic Algorithm. This combination will allow us to find the limits which, with the minimum number of inputs, provide the minimum error in the estimation.

As a different way to improve our model, and after some analysis of the predicted load compared to the real load of the system, we finally propose using piecewise functions. This technique will allow us to reduce the error of the model generated when it is grouped around a concrete CPU load. Therefore, we can generate multiple models. The selection of the model to be used is done automatically according to the input value of the histogram.

We have already stated that we will be using *LASSO* and *GA* for obtaining the CPU models. For a better understanding of our work we provide a brief

background on this techniques.

6.1.2.1 Least absolute shrinkage and selection operator (LASSO)

Tibshirani proposes the least absolute shrinkage and selection operator algorithm (*LASSO*) [171] as an algorithm that minimizes residual summation of squares according to the summation of the absolute value of the coefficients that are less than a constant.

The algorithm combines the favorable features of both subset selection and ridge regression like stability, and offers a linear, convex and derivable solution. *LASSO* provides interpretable models shrinking some of the coefficients and setting others to exactly zero values for generalized regression problems.

For a given non-negative value of λ , the *LASSO* algorithm solves the following problem:

$$\min_{\beta_0, \beta} \left(\frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \beta)^2 + \lambda \sum_{j=1}^p |\beta_j| \right) \quad (6.1)$$

where:

- β : vector of p components. *LASSO* algorithm involves the L^1 norm of β
- β_0 : scalar value.
- N : number of observations.
- y_i : response at observation i .
- x_i : vector of p values at observation i .
- λ : non-negative regularization parameter corresponding to one value of Lambda. The number of nonzero components of β decreases as λ increases.

6.1.2.2 Genetic Algorithms

Genetic Algorithms (GAs) [172] are a method for solving optimization problems based on natural selection. A GA requires a population of candidate solutions (individuals or chromosomes), coded with a genotype. The genotype is a list of integer values, that are called genes, that can be translated to the desired solution format, which is the phenotype. With a fitness function the algorithm calculates the quality of every individual solution of the population. The GA starts with

6. Attacks

a population of individual solutions to the problem. It iteratively creates a new generation, a set of new candidate solutions, from the previous one with three operators:

- Selection: the candidates that best fit the problem are selected for the next generation (lower value of the fitness function).
- Crossover: two candidate solutions (parents) are selected between the best candidates to generate a new candidate by mixing their chromosomes. The child is generated with sequence of genes from both parents.
- Mutation: altering one or multiple genes of a chromosome to generate a new candidate solution.

The configuration of the GA includes the genotype, the fitness function definition, the number of individuals in each generation, the fraction of new candidate solutions generated with each operator and the number of generations to be executed to find the best solution.

The goal of our GA is to reduce the number of input features of the model generated using *LASSO*. The candidate solutions represent different histogram configurations, considering the width of each time interval. That is, the population will be composed of different histograms, which are in turn based on the original measurements.

Each of the new histograms generated by the GA, is then considered as an input for for the *LASSO* that computes the coefficients and the independent term in the linear model. Once the linear model has been obtained, we get the error of each estimation for each input distribution.

As a result, our GA+*LASSO* framework solves our optimization problem that targets the generation of accurate CPU load models for host machines from execution time histograms with a compromise between the accuracy of the model and number of samples used as input. We provide further details about our actual configuration of the algorithms in the following sections.

6.1.3 Experimental Setup

In this section we describe the scenarios in which we have performed our experiments, and detail the processes of gathering the data that will be used for obtaining our models, and the concrete techniques employed.

6.1.3.1 Scenarios

We consider two different scenarios to evaluate our approach: virtualized and non-virtualized environments. For each of the scenarios we run our experiments in three different intel machines, each with different number of cores:

- Intel Xeon CPU E3-1226 v3 (3.30GHz), 4 cores.
- Intel Core i7-4790(3.60 GHz), 4 cores with two threads per core.
- Intel Xeon CPU E5-2620 (2.00GHz), 6 cores with two threads per core.

The OS of all the machines is Centos, and the hypervisor for all our experiments in virtualized environments is KVM. In all our VMs we have also Centos 7.0 installed. Each of the instances has 2 virtual CPUs (VCPUs), although only one would be enough as we are only running one process, the encryption process we measure. We have selected machines with different number of cores, to evaluate to which extent the model can be generalized. Since our test process consumes high amounts of CPU, the minimum load observed in our samples is the load we generate with our test process, which is around $(1/Numberofcores) \times 100$. That is, our test process uses almost one core during its execution.

6.1.3.2 Data acquisition

In both scenarios (non-virtual and virtual), the way of collecting the data is similar. There are two main processes involved in the procedure of gathering the data we use to train and test our models:

- The process we monitor to evaluate the effects of the load on it. It consists in an AES encryption operation, from the OpenSSL library. During a fixed period of time, one second in our experiments, we perform one operation after another and measure the time it takes to perform each one. To obtain the times we use the `rdtsc` instruction, which returns the current value of the processor's timestamp in cycles and is available inside the VMs. At the end of each period, we have an array with the times of each encryption performed and a counter with the number of encryptions that have been performed. We construct an histogram with the number of encryption operations whose duration is between two time values separated by 20 CPU cycles, starting in 100 cycles and ending at 1700 cycles. The total number of time divisions is consequently 80, which are the inputs of our models. Additionally, we store a timestamp tag with the previously computed histogram, obtained with the function `gettimeofday`.

- The process in charge of collecting information about the real use of the host. We collect from the host the values of the CPU load by reading the `/proc/stat` file once a second. We obtain the busy and idle cycles of each core of CPU in the last time window by subtracting the previous value from the current one. Then, we calculate the CPU utilization as the percentage between busy cycles and total cycles. As in the previous case, we store the value of the CPU utilization with its corresponding timestamp. We use the function `gettimeofday` to ensure that both the file with the histograms and the file with the real CPU load can be aligned. The function `rdtsc` could only be used for the non-virtualized environment.

Additionally we generate different CPU loads in the host using `lookbusy` [173], a synthetic load generator. `Lookbusy` is an application for generating load on a Linux system. It makes a system as busy as specified. Load can be induced at a fixed level or on a repeating cycle, allowing the definition of different load profiles. For example, it is possible to imitate a daily traffic load curve in a server. The `lookbusy` application adjusts its own consumption up or down to compensate other loads on the system.

Once we have all the data, and before using it as an input for the regression algorithm, we have to match each sample of the histogram with its corresponding value of the real CPU load. To do so, we compare the timestamps of the data collected with the two processes and assign each histogram with a load value if their timestamps are close enough. At this point we have a file with 81 entries, 80 that contain the samples in each bin, and the remaining one the value which is expected to be the output of our model, that is, the real load.

6.1.4 Methodology

6.1.4.1 LASSO

We use the Matlab software [174] to perform the regression using *LASSO* algorithms. We call the function `lasso(X,Y,'CV',10)` where the predictors in `X` are the values of the histogram and the responses in `Y` are the values of the CPU load. With this call, we also indicate Matlab to use K-fold cross validation to estimate the MSE of our model, and the K value is set to 10. As a result of this call, we obtain the coefficients of the fit, and their associated errors. In other words, we have our model of the host CPU load as a function of our encryption times histogram. One example of the results obtained, can be seen in Figure 6.3. Figure 6.3 also shows that the number of required inputs to compute our model is high, while the error rates are reasonable. This is the reason why we decided to tune our inputs with a GA.

6.1. Inferring the CPU load of a host from a process running on it

(a) Coefficients fit by LASSO

(b) MSE error

Figure 6.3: Outputs of the *LASSO* regression (10-fold cross validation)

6.1.4.2 LASSO tuned with GA

The Matlab software also includes a toolbox that allows the use of customized GA. We have used this toolbox to define our population individuals or phenotypes, a fitness function and the mutation and crossover functions which will be used to obtain the next generation.

Population: we consider 100 individuals per generation. The chromosomes of each individual represent the width of the time divisions of the histogram. With this data we are able to reconstruct our histograms as if the time limits were placed in different positions.

Fitness function: The objective of a GA is to find an individual that minimizes the output of its fitness function. We propose a fitness functions composed of 2 different equations, which are selected depending on the value of the MSE. We use as a MSE reference the error obtained when fitting the model with the *LASSO* with the histogram unmodified. iFor each individual, the fitness function compares the MSE obtained after performing the *LASSO* with the recalculated histograms to the reference MSE. If the MSE is in the range of the reference MSE ($< 20\%$), the equation used in the fitness function is the MSE plus a penalization term given by the number of the significant variables of the evaluated model is used. If the MSE is over the MSE reference range ($geq 20\%$), the equation is the

6. Attacks

calculated MSE multiplied by the number of significant variables of the model. This way, the best individuals are those whose MSE is similar to the original with the minimum number of inputs.

Next generation: the next generation is obtained as follows: 5 individuals are the "elite" and remain unchanged, the roulette wheel selection algorithm selects the parents for the next generation, 80 children will be obtained through crossover and the remaining children through mutation. In the crossover step, two parents are combined to generate one child, we randomly select one point of the phenotype and "cut" both parents at it. Next, the child is obtained recombining one part of each parent. We define a mutation rate of 1%. That is, each child obtained through mutation has only one gen, also randomly selected, different than its parent.

The GA is executed during 500 generations, which takes a mean time of about 3.5 hours for each model. In the following section we present the results for each approach.

6.1.5 Results

We present a table summarizing the results in terms of MSE, standard error (SE) and number of significant features in non-virtualized and virtualized environments, for each of the two approaches: *LASSO* and *LASSO* tuned with GA. Instead of showing all the results together, we present the results for each of the machines individually.

Table 6.1: MSE, SE and number of features in non-virtualized and virtualized environments for the Xeon E3

	Non-virtualized			Virtualized		
	MSE	SE	Features	MSE	SE	Features
Lasso	9.22	0.206	49	8.59	0.144	61
Lasso + GA	10.48	0.065	1	9.89	0.215	2

Table 6.1 includes the results for the Xeon E3 machine. At this point we are able to provide the CPU load estimation as a function of the histogram weights h_i . Note that each weight refers to the new bins calculated with the GA, so h_i stands for the i th bin of the new distribution. As a result, we can estimate the load as:

$$CPULoad(Nonvirt) = 27.6 + 39.45 * h_5 \quad (6.2)$$

$$CPULoad(Virt) = 103.45 - 2.24 * h_1 + 35.15 * h_3 \quad (6.3)$$

In this situation, the GA has been really useful to simplify our model. The

6.1. Inferring the CPU load of a host from a process running on it

errors are slightly bigger, but not statistically significant. At this point, we decided to exchange the models and use the model obtained for the non-virtualized environment to estimate the CPU in virtualized environments. As a result we obtained a MSE of 51.12 in the estimation. However, even in this situation, we are able to detect changes in the CPU load. This means that with this model will still be useful to detect the activity of neighboring processes. Thus, if we have access to a machine similar to the any of the machines used by cloud providers we could, in theory, use the model retrieved for that machine to calculate the load of that host.

Table 6.2: MSE, SE and number of features in non-virtualized and virtualized environments for the i7 processor

	Non-virtualized			Virtualized		
	MSE	SE	Features	MSE	SE	Features
Lasso	4.19	0.037	51	4.47	0.009	59
Lasso + GA	4.53	0.026	6	8.39	0.411	2

The results for the i7 machine can be found in the table 6.2. In this case, the respective equations for CPU load estimation in the i7 machine are:

$$CPULoad(Nonvirt) = 71.93 - 0.65 * h_2 + 0.63 * h_4 + 0.78 * h_5 + 6.72 * h_6 - 0.38 * h_8 + 0.62 * h_{10} \quad (6.4)$$

$$CPULoad(Virt) = 60.08 - 0.46 * h_2 + 120.95 * h_4 \quad (6.5)$$

In this case, there is a noticeable difference between virtualized and non-virtualized environments. In both scenarios the number of input features required to estimate the load of the host has been reduced significantly by using the GA. However, in virtualized environments, the error has increased from 4.47 to 8.39 while in non-virtualized environments it remains similar. We have also tried to interchange the models. In like manner, we obtained a higher MSE (58), but we are still able to notice changes in the CPU load.

Table 6.3: MSE, SE and number of features in non-virtualized and virtualized environments for the Xeon E5 machine

	Non-virtualized			Virtualized		
	MSE	SE	Features	MSE	SE	Features
Lasso	27.79	0.565	56	37.76	0.327	76
Lasso + GA	33.24	0.112	6	56.61	1.12	4

Finally, the results referred to the Xeon E5 machine are presented in the table

6. Attacks

6.3. The resulting models for CPU load estimation for the Xeon E5 machine are:

$$CPULoad(Nonvirt) = 16.34 - 0.25 * h_2 + 4.18 * h_3 + 1.55 * h_6 + 0.43 * h_7 + 1.56 * h_8 - 81.46 * h_9 \quad (6.6)$$

$$CPULoad(Virt) = 71.45 - 0.65 * h_2 - 0.60 * h_4 + 0.03 * h_5 + 1757.6 * h_7 \quad (6.7)$$

This Xeon E5 machine is the one which obtains the worst results, in both virtualized and non-virtualized environments. The main difference between this machine and the remaining ones (apart from the number of cores) is that it had a different governor managing the frequency of the cores. For this reason we had one background process collecting information about the frequency of the CPU. We have observed several changes in the frequency while we were gathering the data for our models. These results suggest that it may be interesting to include the frequency in our models.

Figure 6.4: CPU real and estimated load for the Xeon E3 machine

Figure 6.4 shows the values of the real and estimated CPU load of the host for a virtualized environment on the Xeon E3 machine. This data has been collected in the runtime, we have implemented the equation that gives the CPU in the program that computes the histograms. Consequently, this belongs to a different experiment and is not the same data used for training our models. The reader can observe that both plots look quite similar. However, if we look at the figure more carefully, we can see that our estimated load is unable to reach the minimum load of the host. This suggests that, if we are able to distinguish between the situation when there are other processes executing from the opposite scenario, we could build a different model. In fact, it is possible to distinguish both situations easily in the Xeon E3 machine. We have determined an empirical threshold taking into account only the values of h_5 and h_3 . When these values are below the threshold, the test process is "alone". We have used the real load

6.1. Inferring the CPU load of a host from a process running on it

associated with this values to train a new model using the *LASSO* that in turn consists of two models. The results for our new models are:

$$CPULoad(Nonvirt) = \begin{cases} 24.85 + 53.11 * h_5 & h_5 < 0.08 \\ 27.6 + 39.45 * h_5 & other \end{cases}$$

$$CPULoad(Virt) = \begin{cases} 124.75 - 3.42 * h_1 + 28.02 * h_3 & h_1 > 31 \\ 103.45 - 2.24 * h_1 + 35.15 * h_3 & other \end{cases}$$

These piecewise models reduce the MSE in these case up to 9.37 and 8.07 respectively for non-virtualized and virtualized environments. We have tried to obtain similar functions for the other processors, but we have not been able to find a proper threshold to distinguish when to use each model. We have presented a discussion about the utility of having accurate models to estimate the CPU load of a server in cloud computing environments. We propose to detect co-allocation of virtual machines (VMs) by estimating the host CPU load and triggering an event in the victim VM. A synchronized variation in the load of the host with the trigger can be used as indicator of both VMs being executed in the same physical machine.

Co-residency detection is not the only application for our models. Now that we have proved that it is possible to derive the CPU load of a server by measuring the time it takes a process to execute n times during a fixed period of time. We can use this information to create a covert channel between machines. For example, one machine can send information to other just by generating a sequence of execution or not-execution to send a 1 or a 0. Note that our model has been trained using an interval between samples of 1s. In this case, the maximum capacity of this channel is 1 bit per second.

To sum up, the histogram with the number of executions for multiple time intervals is enough to obtain a reasonable predictor of the CPU load in the host. The use of the Genetic Algorithm greatly simplifies the process of selection of a suitable model. The resulting model is a linear formula that needs few inputs of the histogram to give an estimated load of the host. Note that, although we have selected an AES encryption as the reference process, any relatively short process that uses all the available CPUx, could similarly be used to compute the models. We believe that our models can be used in a public cloud, as part of a co-residency detection algorithm or to create a cover channel. However, these models are machine-dependent, so we would require access to a machine similar to the machine we want to monitor in the cloud.

6.2 Improving traditional attacks against AES

There are different ways to improve the efficiency of a cache attack. In this section we propose to change the approach to process the retrieved data in existing cache attacks, considering both the algorithm and the cache at the same time, to drastically reduce the number of samples required to gain information on the AES key. We explore cache miss information instead of cache hits. Since this approach is general, all the attacking techniques proposed so far (FLUSH+RELOAD FLUSH+FLUSH PRIME+PROBE and PRIME+ABORT) can benefit from it. These cache attacks just require the victim and the attacker to share the cache to recover fine grain information. Researchers have also become concerned that these attacks represent a serious threat. For this reason, many approaches have been proposed that try to avoid the leakage [120, 121, 122] or to detect attacks [156, 67, 66, 69]. Considering that, to the best of our knowledge, no hardware countermeasure is implemented in real cloud environments and no hardware manufacturer has changed the cache architecture, an attacker wishing to exploit side-channel cache attacks will only have to take into account the countermeasures based on detection that can be implemented by users. In that scenario, reducing the number of samples required to complete the attack, can be the difference between success and failure.

Traditionally, cache attacks against AES use the information about the victim access to a memory address (cache hit). In contrast, we show that using non-access provides much more information, and we demonstrate that the power of cache attacks has been underestimated during these last years. Using cache misses as the source of information, we are able to retrieve the 128-bit AES key with a reduction in the number of samples between the 93% and the 98% compared to the traditional approach. Furthermore, this attack is adapted and extended in what we call the encryption-by-decryption cache attack or EBD, to obtain a 256-bit AES key. In the best scenario, this approach obtains the 256 bits of the key of the OpenSSL AES T-table-based implementation using less than 10000 samples; that is, 135 milliseconds, proving that AES-256 is only about 3 times more complex to attack than AES-128 via cache attacks.

This part of the work considers one more time the T-table-based implementation of AES. As stated in previous sections, this implementation is known to be vulnerable to cache attacks. However, it is commonly used for comparison, and to demonstrate different attack techniques. In this work, it serves to the purpose of showing the accuracy of the information gained from the non-access to memory (*cache misses*) and to quantify the improvement that this approach represents compared to the traditionally used approach based on access. The approach is proved in both virtualized and non-virtualized environments.

We are able to improve the results of previously published side channel attacks decreasing the number of samples required to retrieve the whole key by multiple orders of magnitude. Moreover, the presented approach is less prone to consider false positives memory access. Regarding non-access, a false positive is considered if the data has been loaded into the cache and removed from the cache in a short period of time, which is unlikely to happen. Whereas a false positive in the traditional approach is considered whenever the data is loaded into the cache in any other round of the AES encryption rather than the target round, or whenever this data is predicted to be used, so it is speculatively loaded into the cache, being both frequent options.

In section 3.2.1 we described the main attack techniques we are going to exploit. We also explained the requirements for each attack in that section, so we refer the reader back to that section and to the background section for further details about the microarchitecture. However, we consider it necessary to give more information and details than the ones provided in section 5.1.1 about AES to fully understand the attack.

6.2.1 Attack on AES

In the following, we describe the fundamentals of AES encryption and decryption algorithms. Although the algorithm has already been introduced in this work, we include in this section further explanation, so the reader becomes familiar with the notation used and to ease the understanding of the attack. We explain the insides of the attacked T-table-based OpenSSL AES implementation. We present the non-access cache attack against AES-128, which outperforms previously published cache attacks. Later, we explain how the approach followed in this non-access attack can be extended to perform a practical attack on AES-256. We name it Encryption-by-Decryption Cache Attack, as we use the information from the encryption to obtain a decryption round key, which can be transformed into an encryption round key. This way, we are able to obtain information from two different and consecutive encryption rounds.

6.2.1.1 AES fundamentals

The AES algorithm is explained in Section 2.7.1. The encryption process is depicted in Figure 2.9. The T-Table implementation is vulnerable, because it performs memory key dependent access. In particular, the first and last round of such AES implementation are slightly different than the others. They consist of an XOR operation between some values of the lookup T-Tables and the respective round key. The target of the attack is the last round of AES, as Irazoqui et al. did[31]. Each output ciphertext byte can be summarized as seen in Eq. 6.8; k_i is

6. Attacks

the i byte of the round key, $Te_{(i+2)\%4}$ is the lookup table corresponding to the byte and s_i is the state byte i from previous round, which is used as index of the lookup table. Gaining information on the data used from the T-tables, with a known output, provides information on the last round key (which is straightforward related to the encryption key).

$$C_i = k_i \oplus Te_{(i+2)\%4}[s_i] \quad (6.8)$$

In order to obtain an AES-128 key, an attacker must guess one round key. Considering AES-256, at least 2 consecutive round keys are required. AES arranges both round key and the data in form of a 4x4 matrix of byte elements. If we follow the terms used in [25], we can denote $S^r, 0 \leq r \leq N_r$ the input state to round r . Each of its byte elements is then denoted as $S_{i,j}^r, 0 \leq i < 4, 0 \leq j < 4$. i indicates the row of the state and j the columns. We use only one subindex j to refer to a column of the state S_j . Considering a plaintext block $p_0p_1p_2\dots p_{15}$, the elements of the initial state S^0 of the algorithm are $S_{i,j}^0 = p_{i+4j}$.

The normal round transformation in the encryption process consists of four steps denoted: **SubBytes**, **ShiftRows**, **MixColumns** and **AddRoundKey**. **SubBytes** is the only non-linear transformation. All these steps have an inverse step, which are used in the decryption process: **InvSubBytes** (using S-box S_{RD}^{-1}), **InvShiftRows** (rotating to the right), **InvMixColumns** (multiplying by the inverse matrix) and **AddRoundKey**. The normal round transformation is applied in all but the first and last rounds. The first round applies only the **AddRoundKey** step. The last round is a normal round without the **MixColumns** step.

The decryption algorithm can be implemented by applying the inverses of the steps in a reverse order, being **InvSubBytes** the last step of each round excepting the last one. It is preferable to perform the non-linear step first in typical implementations. In [25] they present the *Equivalent Decryption Algorithm*, with a sequence of steps equal to the encryption, by switching the order of application of **InvSubBytes** and **InvShiftRows** (the order is indifferent) and **InvMixColumns** and **AddRoundKey**. The round key of all but the first and last rounds needs to be adapted for this purpose, by applying the **InvMixColumns** step to it.

The **key** fact to understand the attack against AES-256 is that the state S^r can be reached in the r round of the plaintext encryption process or in the $N_r + 1 - r$ round of the ciphertext decryption process.

6.2.1.2 AES T-table implementation

The steps of the round transformation include a non-linear S-box substitution and a matrix multiplication. In order to reduce the number of operations performed

during each round and to improve performance, AES software implementations without special assembly instructions use tables (T-tables) with precalculated values for each of the possible 256 values of an input element. In this work, we evaluate the T-table-based implementation available in OpenSSL version 1.0.1f or newer versions when compiled with the no-asm flag.

In the aforementioned implementation, the state is represented by four 32-bit variables starting with s or t , one for each column of the state. The name of the variable starts with s for odd rounds and with t for even rounds, considering the first round to be 0. The Most Significant Byte (MSB) of the variable represents the element in row 0, while the Least Significant Byte (LSB) represents the element in row 3. Encryption and decryption algorithms use 4 tables, each containing the result of S-box substitution and part of the matrix multiplication. The tables start with **Te** for encryption and with **Td** for decryption.

Generically, we denote each of the 4 tables T_i (**Te** $_i$ and **Td** $_i$), where i is the column of the multiplication matrix considered in the table. Table T_i contains the contribution of the element in row i to each row of the resulting column state. That is, each of the 256 positions of the table (2^8) contains a 32-bit value, where each byte is the contribution to each row, aligned to perform the addition of the different contributions. The size of each table is 1024 bytes. The column state is calculated by adding (bitwise XOR in $GF(2)$) the contributions from the different rows and the round key. Therefore, the four tables are accessed for each output column state. An appropriate selection of the input variable s_j for the different rows is used to perform the **ShiftRows** step; for the destination variable t_j , variable s_{i+j} is used for row i (index of table T_i). In this example, and for the entire work, the addition in subindexes i and j is done modulus 4. For example, in round 1 we can obtain t_0 as follows:

$$t_0 = Te_0[s_0 \gg 24] \oplus Te_1[(s_1 \gg 16) \& 0xff] \oplus Te_2[(s_2 \gg 8) \& 0xff] \oplus Te_3[s_3 \& 0xff] \oplus rk[4];$$

The last round does not include the matrix multiplication. A table for S_{RD}^{-1} (only **InvSubBytes** step) is required for decryption algorithm (table **Td** $_4$). As there are coefficients with value 1 in the multiplication matrix of step **MixColumns**, the table is not needed for the encryption algorithm. It is implicitly available in the **Te** tables. In this case, **Te** $_{(i+2) \bmod 4}$ is used for row i , because it has coefficient 1 for that row.

6.2.1.3 Attacking AES

As it can be inferred from the previous description of the T-table implementation, the access to each **Te** $_i$ depends on the key. Thus, cache hits and misses depend on the key. In the following section 6.2.1.4 we explain how to retrieve the 128

6. Attacks

bits key using information about cache misses and the ciphertext.

We show that using cache misses we are not only able to recover an AES 128 bit key using significantly fewer samples than previous approaches, but also we can recover the 256 bits of an AES key. To this end, we need to recover information about two consecutive rounds. As we explain later, we perform this attack in two steps. The first one targets the last round and the second one the penultimate round. A direct cache attack to the penultimate round key is not feasible, so we use the equivalences between encryption and decryption to transform the data and to be able to use cache misses to retrieve the key.

When describing the attacks, we use a nomenclature which is consistent with the terms used when describing AES. The iteration variables, subindexes and superindexes follow this rules:

i is used to refer to the rows of the state or round key. As a subindex, it indicates the row of the element. As an iteration value, it is used for accessing elements related to a row (the T-tables).

j is used to iterate the columns of the state or round key. As a subindex, it indicates the column of an element.

t is the iteration variable for traversing the different encryptions of the experiment.

l is the iteration variable for traversing the values of a Cache line.

r is used as superindex to indicate the round of a variable (state, round key).

The elements used in the algorithm are represented by capital letters.

S^r represents the state of an AES encryption algorithm before round r . The 16 bytes of the state can be accessed using i and j subindexes. The state is different for each encryption performed, and the array $\overline{S}^r$ represents the whole set.

D^r is the equivalent to S for the decryption algorithm.

K^r represents the encryption round key for round r . Each of the 16 bytes can be accessed using i and j subindexes.

Kd^r represents the decryption round key for round r . It can be calculated from the corresponding encryption key K^{N_r+1-r} by applying the `InvMixColumns` step.

X_i is information on access (1) or non-access (0) to a subset of table Te_i . The subset contains 16 values (64 bytes), which is the cache line length. $\overline{X_i}$ represents the array containing the access information for each encryption performed in the experiment.

CK^r is a set of arrays with 256 positions (the possible values), one for each round key byte. Each position represents the amount of discarding votes received by the candidate for that concrete key byte. The position with the minimum value after the analysis (we call the function **argmin**) is the correct key byte value. Its elements can be accessed using i and j subindexes.

CKd^r is the equivalent to CK^r for a decryption round key.

6.2.1.4 Attack on AES-128

We assume that the attacker shares the cache with the victim, which means he can monitor access to the cache with line granularity. We also assume that the attacker has access to the ciphertext. The attacker needs to recover information from each of the four T-tables. To do so, he can monitor one line of one table, one line of each T-table or even try to monitor the 16 lines of each of the 4 tables at a time (64 lines). The number of samples required to retrieve all the bits of the key will vary in each case as will do the effect of the attack observed by the victim [156]. The more lines monitored at a time, the more noticeable the attack will be. While a slight increase in the encryption time can be assumed to have been caused by other programs or virtual machines running on the same CPU, a higher increase in the encryption time would be quite suspicious.

For each of the evaluated techniques, i.e. PRIME+PROBE PRIME+ABORT FLUSH+RELOAD and FLUSH+FLUSH we use a similar process to retrieve the desired information. The process followed during the attack involves three steps and is the same in all cases:

1. **Setup:** Prior to the attack, the attacker has to retrieve the necessary information about where the T-tables are located in memory; that is, their virtual addresses or the cache set in which they are going to be loaded. This way he ensures that the cache is in a known state, ready to perform the attack.
2. **Measurement collection:** In this step the attacker monitors the desired tables, applying each technique between the requested encryptions and stores the gathered information together with the ciphertext. As explained in section 3.2.1, this step can be subdivided into three stages: **initialization**, **waiting** and **recovering**.

6. Attacks

- 3. Information processing:** Finally, the attacker uses the information recovered in the previous steps and the information about the T-tables (the contents of the monitored line) to retrieve the secret key.

These main steps are equivalent to previous proposals [31, 79]. Other proposals are also applicable to the non-access attack, as the one suggested by Gülmezoglu et al. [56] which aims to detect the bounds of the encryptions instead of triggering them. However, this approach also depends on the accuracy of the detection, which introduces a new variable to be considered when comparing the effectiveness of the attack approaches. In contrast, the aforementioned setup can be generalized for all the considered techniques, allowing a fair comparison.

In the following we describe each of the steps for each of the considered algorithms and their particularities in the experimental setup. We consider two scenarios. In the first one, the attack is performed from a spy process running in the same OS as the victim. In the second one, the attacker runs in a VM and the victim runs in a different VM and they are both allocated in the same host. Table 6.4 includes the details of the machine in which the experiments are performed. For the cross VM scenario we use KVM as hypervisor and the OS of the instances is CentOS 7.6 minimal. The steps are equal in the two considered scenarios.

Table 6.4: Experimental platform details.

Processor	Cores	Frequency	OS	LLC slices	LLC size	LLC ways
Intel core i5-7600K	4	3.8 GHz	CentOS 7.6	8	6 MB	12 ways

Setup

The prerequisites and the procedure that an attacker must follow to successfully perform FLUSH+RELOAD and FLUSH+FLUSH are different than the required ones for PRIME+PROBE and PRIME+ABORT. For instance, since *FLUSH+Something* attacks rely on shared memory, an attacker wishing to exploit this feature needs to do some reverse engineering on the library used by the victim to retrieve the target virtual addresses. We have used the *gdb* debugger to that end. Note that the offset between the addresses of the beginning of the library and the target symbol (the table) is constant. Thus, once this offset is known, the attacker can easily get the target address by adding this offset to the virtual address where the library is loaded.

In contrast, *PRIME+Something* attacks require the physical address of the target; more accurately, it is necessary to know the set and the slice in which the data will be loaded. This information can be extracted from the physical

address. Part of the physical address can be directly inferred from the virtual address. Indeed, the bits of the address that points to the elements within a cache page are part of their physical address. For example, if the size of a page is 4 KB, the 12 lowest significant bits of the address will keep the same when translating it from virtual to physical. However both virtual and physical addresses of the data that the victim is processing are unknown to the attacker. To overcome this difficulty, an attacker needs to create its own eviction set (a group of W elements that map exactly to one set) and profile the whole cache looking for access being carried out by the victim.

Instead of recovering the complex address function of a processor, we create the eviction sets dynamically using the technique summarized in the Algorithm 1 of the work of Liu et al. [71]. We have also enabled *hugepages* of 2MB in the system to work with 21 known bits. Since the number of cores of the processor is a power of two, we only need to obtain the mapping of one of the sets for each slice (the function that determines the slice in which the data is going to be placed only uses the *tag* bits). The remaining sets that will be used during the profiling phase and the attack itself can be constructed using this retrieved mapping.

Measurement collection

The measurement collection process is somehow similar for all the considered techniques. Algorithm 7 summarizes the general method employed to acquire the necessary information about the victim access to the tables. This algorithm considers that one element of each table is being monitored. Later, this information will be used to obtain the full key.

In the initialization, the attacker has to evict the data of one line of each Te_i table from the cache. That is, the attacker ensures he knows the state of the cache before the victim uses it. Since each cache line holds 64 bytes, and each entry of the table has 4 bytes, it holds 16 of the 256 possible values of the table. Then, in the waiting stage, he triggers an encryption. Once it has finished, in the recovering stage, the attacker checks the state of the cache. If the cache state has changed, this means that it is likely that the victim had used the data of that T-Table. The different approaches considered in this work differ in the way they evict the data from the cache during the initialization and in the way they recover the information about utilization of the evicted table line.

In the initialization stage of *FLUSH+Something*, the target lines are evicted from the cache using the *clflush* instruction available in Intel processors. In *PRIME+Something*, the attacker evicts the target lines by accessing the data of the created eviction sets that map to the same region as the target lines. Although we have already revealed the Intel replacement policy and for comparison with

6. Attacks

Algorithm 7 Generic attack algorithm for cache attacks against T-Table AES implementations

Input: $\text{Address}(\text{Te}_0), \text{Address}(\text{Te}_1), \text{Address}(\text{Te}_2), \text{Address}(\text{Te}_3)$
1: $\triangleright$ Addresses of the T-Tables
Output: $\overline{X}_0, \overline{X}_1, \overline{X}_2, \overline{X}_3, \overline{S}^{N_r+1}$ $\triangleright$ Information about the access and ciphertext
2: **for** $t = 0$ **to** $\text{number_of_encryptions}$ **do**
3: **for** $m = 0$ **to** 4 **do** $\triangleright$ INITIALIZATION
4: **Evict from cache** (Te_m); $\triangleright$ The attacker prepares the cache for the attack
5: **end for**
6: $\triangleright$ WAITING
7: $\overline{S}^{N_r+1}[t] = \text{encrypt}(\text{random plaintext});$ $\triangleright$ The victim performs one encryption
8: **for** $m = 0$ **to** 4 **do**
9: **Infer victim access to** (Te_m) $\triangleright$ RECOVERING
10: **if** $\text{hasAccessed}(\text{Te}_m)$ **then** $\triangleright$ The attacker Reloads, Flushes, Probes the target or gets the Abort
11: $\overline{X}_m[t] = 1;$ $\triangleright$ The attacker decides if the victim has used the data.
12: **else**
13: $\overline{X}_m[t] = 0;$
14: **end if**
15: **end for**
16: **end for**
17: **return** $\overline{X}_0, \overline{X}_1, \overline{X}_2, \overline{X}_3, \overline{S}^{N_r+1};$

previous works we assume that Intel implements a pseudo LRU eviction policy and replicate other attacks exactly. As a result, accessing W elements that map to a set implies that any older data in the cache will be replaced with the attacker's data. In PRIME+ABORT this operation is performed inside a transactional region defined using the Intel TSX extension.

In the recovering stage of FLUSH+RELOAD the attacker access each of the evicted lines measuring the access times. If the time is below a threshold, it means the line was in the cache, so the algorithm must have accessed it in any of the rounds. FLUSH+FLUSH decides if the data has been used by measuring the time it takes to flush the line again. The flushing time depends on the presence of the line on the cache; it takes longer to flush data that is located in the cache. The PRIME+PROBE recovers information on access by accessing the data from the eviction set and measuring the access time.

As recommended in previous works [71], we access the data within the eviction set in reverse order in the Probe step. That is, the eviction set is read following a zig-zag pattern. If the data of the table is used by the encryption process, part of the attacker's data will be evicted; thus, the time it takes to “Probe” the set

will be higher. In the PRIME+ABORT as the eviction set was accessed inside a transaction, any eviction of the attacker's data from the cache causes an abort. The attacker defines a handler for this abort that evaluates its cause. There are different abort causes which are identified with an abort code [20]. This abort code is loaded into the *eax* register and it is read in the handler function to check if the abort was due to an eviction.

Figure 6.5 shows the distribution of the measured times during the Reload step (Figure 6.5a) and during the Probe step (figure 6.5b). Note that it is easy to establish a threshold to distinguish between access and not-access to the monitored Te table. Also note that there are multiple peaks in the PRIME+PROBE plot. Such peaks are due to the replacement policy combined with the access pattern. The zig-zag access pattern implies that some elements are in the low level cache so their age is not updated. Therefore, such elements may be evicted in lieu of the target.

Figure 6.5: Timing information obtained when attacking AES in the Reload and Probe steps in the test machine. The threshold is shown in red. For the Flush+Reload attack cache misses are represented by times over the threshold. In contrast for the Prime+Probe attack non-access are below the threshold

Information processing

This final step is common for all the attacks. In this stage the proposed approach differs from previous attack strategies. Traditional approaches for the considered attacks take into account if the data stored in the monitored cache line has been used by the victim. In contrast, we consider the opposite situation, i.e. the victim has not used that data. The proposed approach is based on the following observations:

- Whenever the retrieved information shows that the victim has used the considered data, it could have been used during the last round or during

6. Attacks

any of the remaining ones. Thus, whenever the data is used in a different round, this approach introduces “noise”. On the other hand, if a monitored cache line has not been accessed during the encryption, neither has it been accessed in the last round nor in any other round.

- Speculative execution of pieces of code, or prefetching of the data that the processor believes it is going to be used in the near future, may involve that the attacker sees in the cache data that has not been used by the victim (false positive). Nonetheless, in order to obtain a false positive in the non-access approach, the data has to be used during the last round of the encryption and then flushed away by other process during the short period of time after the end of the encryption and before the recovering step.
- Each non-access sample reduces the key search space discarding up to n^4 key values (n equals to 16 Table entries stored in the cache line which are used to obtain 4 output bytes). Thus, we obtain information from each of the samples. In contrast, when considering access, only when one candidate value towers over the rest, the retrieved information is useful to retrieve the secret key.
- The information obtained this way is more likely to be applicable to all the rounds of an encryption, or at least to the penultimate round, as it refers to the whole encryption, so we can use this information to retrieve longer keys (192 or 256 bits).

We have seen in section 6.2.1.2 that each table **Te** is accessed 4 times during each round. Therefore, the probability of not accessing a concrete cache line within an encryption is provided by equation 6.9, where n represents the number of table entries a cache line can hold, and N_r the number of rounds of the algorithm. Particularly in this case, where the size of a cache line is 64 bytes, n is 16 (each entry of **Te** has 32-bits) and the number of rounds (N_r) for AES-128 is 10. Consequently, the probability of not accessing the cache line is 7.5%. This means that one of each fourteen encryptions performed gives us useful information.

$$Pr[no\ access\ Te_i] = \left(1 - \frac{n}{256}\right)^{N_r * 4} \quad (6.9)$$

As stated before, we focus on the last round of the encryption process. This round includes **SubBytes**, **ShiftRows** and **AddRoundKey** operations. Referred to the last round input state and the T-tables, we can express an output element by $S_{i,j}^{N_r+1} = K_{i,j}^{N_r} \oplus \mathbf{Te}_{i+2} [s_{i,j+i}^{N_r}]$. When a cache line has not been used during the encryption, we can discard all the key values which, given an output byte, would have had to access any of the **Te**-table entries hold in the mentioned cache line. For example, given a ciphertext byte '0F' and a cache line holding '01' 'AF' 'B4'

6.2. Improving traditional attacks against AES

and '29', if this line remains unused after performing the encryption we discard the key byte values $0F \oplus 01$, $0F \oplus AF$, $0F \oplus B4$ and $0F \oplus 29$. As we have seen in Section 6.2.1.2, the same table is used to obtain all the elements of the row of the output state. This means each Te_i table also discards the $K_{i,j}$ values with the same i value independently of the j index.

Algorithm 8 Recovery algorithm for key byte $K_{i,j}^{N_r}$

Input: $\bar{X}_0, \bar{X}_1, \bar{X}_2, \bar{X}_3, \bar{S}^{N_r+1}$ $\triangleright$ Information about the access and ciphertext collected in the previous step.

Output: $\text{CK}_{i,j}^{N_r}$ $\triangleright$ Secret key values

- 1: **for** $l = 0$ **to** 256 **do**
- 2: $\text{CK}_{i,j}^{N_r}[l] = 0;$ $\triangleright$ Initialization of the candidates array
- 3: **end for**
- 4: **for** $t = 0$ **to** *number_of_encryptions* **do**
- 5: **if** $\bar{X}_{i+2}[t] == 0$ **then**
- 6: **for** $l = 0$ **to** n **do** $\triangleright n$ stands for the number of entries that a cache line holds
- 7: $\text{CK}_{i,j}^{N_r}[\bar{S}_{i,j}^{N_r+1}[t] \oplus \text{Te}_{i+2}[l]] ++;$ $\triangleright$ Vote for discard key candidate
- 8: **end for**
- 9: **end if**
- 10: **end for**
- 11: **return** $\text{argmin}(\text{CK}_{i,j}^{N_r});$ $\triangleright$ The candidate with the fewest votes for discard is the **secret key**.

A general description of the key recovery algorithm for the last round key is provided in algorithm 8. Key bytes of the last round, $K_{i,j}^{N_r}$, are recovered from the output state $\bar{S}^{N_r+1}$ (ciphertext) and the information about the access to each table Te_i , which is stored in $\bar{X}_i$ and was recovered using the algorithm 7. The output state ($\bar{S}^{N_r+1}$) is obtained by arranging the known ciphertext vector as a matrix as indicated in Section 6.2.1.1. The algorithm first initializes an array of 256 elements for each of the 16 bytes of the key ($K_{i,j}$). This array ($\text{CK}_{i,j}^{N_r}$) will contain the discarding votes for each key byte candidate. The candidate with less negative votes is the value of the secret key.

Figure 6.6 shows the values of one of the 16 $\text{CK}_{i,j}^{N_r}$ arrays which contains the discarding votes each possible key value has received during the attack for a concrete byte of the key. The x-axis represents each possible key value, and the y-axis the number of times each option has been discarded. The minimum is well distinguishable from the rest of possible key values and is highlighted in red. The index with the minimum score represents the secret key. This approach allows us to recover the 128 bits of the key with less than 3000 encryptions on average in the best case (the FLUSH+RELOAD attack). We present further results and provide further details in section 6.2.2.

Figure 6.6: Key byte possible values with their associated discarding scores. The value marked in red represents the correct key value and is a well distinguishable minimum in the plot

6.2.1.5 Attack on AES-256

The attack against AES-256 comes as an extension of the previously explained attack on the last round of an encryption. There are no further assumptions than in the previous case. The setup and measurement collection steps are exactly the same for both attacks, and only the information processing phase is different since it includes an extra step. We first retrieve 128 bits of the key using the algorithm 8 and use the guessed 128 bits of the key and the same information that has already been used (non-access and ciphertext) to derive the remaining 128 unknown bits of the key. We will similarly use information about the non-used cache lines to discard key values until we obtain the key. This is possible because this information refers to the whole encryption and, as previously stated, if a cache line has not been accessed within an encryption, it has not been accessed in any of its rounds.

Note that the probability of non-accessing one line when performing an AES-256 encryption, which is a function of the number of rounds, and is given by equation 6.9, is 2.69% (14 rounds). As a consequence the attack on AES-256 will require more samples in order to successfully guess the whole key. In this case about 1 sample of each 37 carries useful information. As a result, we will need to collect at least the same number of useful samples than in the previous case.

The key to understand the attack, is to understand the equivalence between encryption and decryption. Using the decryption algorithm, we transform the data referring to the encryption process in a way that it is possible to easily derive the round key discarding its possible values. For this reason we call this

approach encryption-by-decryption attack or EBD.

If we analyze the AES encryption process, we see that the output of each round is used as an index to access the T-tables in the next round after a **ShiftRows** operation. Taking into account the valuable information of the unused cache lines it is possible to figure out which are the state values that are not possible before the penultimate round, S^{Nr-1} . That is, if a cache line has not been accessed, all the state values that would have forced an access to this line can be discarded.

Once we know the last round key, it is straightforward to obtain the input to that round (S^{Nr-1}) only taking into account the encryption function. However, because of the **MixColumns** operation which is performed in all the previous rounds (except for the initial one), the certainty about the non-usage of a cache line of one table and the knowledge of the encryption function are not enough to obtain information about the key of these rounds. But, if we use the decryption function and take into account the equivalence between encryption and decryption, we can transform the data so we can derive information about an state of the decryption function which is equivalent to the S^{Nr-1} state in the encryption function.

Figure 6.7 shows with dotted lines the instants where equivalent decryption and encryption algorithms have the same values. According to the figure, we establish a bidirectional relation between an encryption state S^r and a decryption state D^{Nr+1-r} using **SubBytes** and **ShiftRows** steps. The relations are defined in Equations 6.10a and 6.10b. These relations and the knowledge of the values which are not possible in the state S^{Nr-1} give us the values which are not possible in the state D^2 . Since the value of what we call intermediate value IV^1 can be easily obtained from the decryption function, the scenario is similar to the previous one. We can write $D_{i,j}^2$ as follows ($D_{i,j}^2 = IV_{i,j}^1 \oplus CKd_{i,j}^1$).

$$D^{Nr+1-r} = \text{ShiftRows}(S_{RD}(S^r)) \quad (6.10a)$$

$$S^r = S_{RD}(\text{InvShiftRows}(D^{Nr+1-r})) \quad (6.10b)$$

Algorithm 9 shows how to obtain the correct candidates for the round key of decryption algorithm round 1 (the second round). The key of the first decryption round is exactly the same as the key used in the last round of the encryption which was previously obtained using the non-access cache attack and targeting the last round (algorithm 8). As we know both input data (ciphertext) and round key, we can perform the first round of the decryption algorithm and obtain the state of round 1, D^1 . Applying the Td-tables on the state we calculate an intermediate value of round 1, IV^1 . At this point if we perform the **AddRoundKey**, the next step of the decryption algorithm, the result would be D^2 , which can be transformed into the encryption S^{13} state as has just been explained using Equation 6.10b.

Figure 6.7: Relation between encryption, straight decryption and Equivalent decryption.

In the T-table implementation, the state (including S^{N_r-1} or S^{13} for AES256) is used as index to access the **Te** tables. Note that not accessing a cache line holding one entry of the table also indicates which indexes of the table are not being used, which is similar to indicating which values of the state can be discarded. Then, we apply Equation 6.10a to transform the discarded values for state S^{N_r-1} to discarded values for state D^2 . After this transformation, we have information on both sides of the **AddRoundKey** in round 1 of the decryption sequence. We can vote to discard possible values of the decryption round key. We vote to those values which, once transformed the decryption state into the encryption state, would lead to a memory access on the unused lines.

The least voted value for each byte is the decryption round key. Both the encryption and decryption keys are related and one can be obtained from the other applying the **InvMixColumns** operation, or an equivalent operation depending on the direction of the conversion. As a result of this information processing, we

6.2. Improving traditional attacks against AES

Algorithm 9 Recovery algorithm for key byte $\text{Kd}_{i,j}^1$

Input: $\overline{X}_0, \overline{X}_1, \overline{X}_2, \overline{X}_3, \overline{S}^{N_r+1}$ and $\mathbf{K}^{N_r}$ ▷ Information about the access, ciphertext and last round key.

Output: $\text{Kd}_{i,j}^1$ ▷ Secret decryption key values

- 1: $\text{Kd}^0 = \mathbf{K}^{N_r}$
- 2: **for** $l = 0$ **to** 256 **do**
- 3: $\text{CKd}_{ij}^1[l] = 0$; ▷ Initialization of the candidates array
- 4: **end for**
- 5: **for** $t = 0$ **to** *number_of_encryptions* **do**
- 6: **if** $\overline{X}_i[t] == 0$ **then**
- 7: $D^0 = \overline{S}^{N_r+1}[t]$;
- 8: $D^1 = D^0 \oplus \text{Kd}^0$;
- 9: **for** $j = 0$ **to** 4 **do**
- 10: $\mathbf{IV}_j^1 = \mathbf{Td}_0[D_{0,j}^1] \oplus \mathbf{Td}_1[D_{1,j-1}^1] \oplus \mathbf{Td}_2[D_{2,j-2}^1] \oplus \mathbf{Td}_3[D_{3,j-3}^1]$ ▷
- Obtain intermediate value
- 11: **end for**
- 12: **for** $l = 0$ **to** n **do** ▷ n stands for the number of entries that a cache line holds
- 13: $\text{CKd}_{i,j}^1 = [\mathbf{IV}_{ij}^1 \oplus S_{RD}[l]] ++$; ▷ Vote for discard key candidate
- 14: **end for**
- 15: **end if**
- 16: **end for**
- 17: **return** $\text{argmin}(\text{CKd}_{ij}^1)$; ▷ The candidate with the fewest votes for discard is the **secret key**.

know two consecutive round keys; that is, the full AES-256 key. This algorithm could be applied recursively for the whole encryption/decryption process, as the cache information is applicable to the whole encryption.

Using this encryption-by-decryption cache attack on AES-256 we obtain the key with less than 10000 samples in the best scenario (the FLUSH+RELOAD attack). We provide further details about the results in section 6.2.2.

6.2.2 Results

As presented in Table 6.4, the experimental setup includes an Intel Core i5-7600K processor (3.80GHz) with 4 CPUs and two core private L1 caches of 32 KB (one for data and other for instructions), a private L2 cache of 256 KB and, finally, a L3 cache of 6 MB (12 ways) shared among all cores whose line size is 64 bytes. It has CentOS 7.6 installed while the Virtual Machines in the experiments have CentOS 7.6 minimal and both have one virtual core. The hypervisor is KVM. The target of the experiments is the 1.0.1f OpenSSL release which includes the T-Table implementation of AES.

6. Attacks

In the previous section, we presented attacks that work for different lengths of AES keys (128 or 256 bits). These attacks assume the knowledge of unused cache lines to discard possible key values and that the ciphertexts are known by the attacker. We have performed the experiments in two different scenarios (virtualized and non virtualized environments) considering different techniques to retrieve information about the cache utilization (PRIME+PROBE, PRIME+ABORT, FLUSH+RELOAD and FLUSH+FLUSH) and different key lengths. In all cases, one line of each T-Table is monitored. That is, we monitor 4 lines per encryption operation.

Table 6.5: Mean number of samples required to retrieve the whole key in non virtualized environments

	<i>FLUSH+RELOAD</i>	<i>FLUSH+FLUSH</i>	<i>PRIME+PROBE</i>	<i>PRIME+ABORT</i>
AES 128	3000	15000	14000	3500
AES 256	8000	35000	38000	8000

Table 6.6: Mean number of samples required to retrieve the whole key in virtualized environments (cross-VM)

	<i>FLUSH+RELOAD</i>	<i>FLUSH+FLUSH</i>	<i>PRIME+PROBE</i>
AES 128	10000	40000	45000
AES 256	28000	100000	110000

Table 6.5 shows the mean number of encryptions that have to be monitored to obtain the full key in non virtualized environments (the victim and the attacker processes run in the same OS). Similarly, table 6.6 shows the results for virtualized environments (the victim runs in one VM and the attacker in a different one sharing the same host).

When applying the PRIME+ABORT technique, we want to monitor 4 different lines. We need to define 4 transactional regions to distinguish between them, which also means we use more resources. Note that we do not include the results for the PRIME+ABORT technique in virtualized environments. The TSX instructions are virtualized, so it should be possible to perform PRIME+ABORT in a cross-VM scenario. However, with the experimental setup consisting on an attacker preparing the cache, triggering an encryption and waiting for the result, we get too many aborts. Since all of them have similar causes, distinguishing between the aborts which are due to the encryption and the ones which are just noise is harder. For this reason, and since we consider that our point has already been proved, we have not evaluated the data collected with the PRIME+ABORT technique in the cross-VM scenario.

For a fair comparison between the access and the non-access approach, we have also retrieved the key for the access approach considering the key of 128

6.2. Improving traditional attacks against AES

bits. Table 6.7 shows the results for both approaches and the percentage of improvement regarding the number of traces required to successfully retrieve the secret key. In all cases the number of samples is reduced significantly.

Table 6.7: Mean number of samples required to retrieve the whole 128 bits key with the access approach (first column), the non-access approach (second column) and the improvement the latter approach means. (V) means virtualized scenario.

	Access	Non-access	Improvement
<i>Flush+Reload</i>	100000	3000	97%
<i>Flush+Flush</i>	250000	16000	94%
<i>Prime+Probe</i>	190000	14000	93%
<i>Prime+Abort</i>	110000	3500	97%
<i>Flush+Reload(V)</i>	400000	10000	98%
<i>Flush+Flush(V)</i>	1120000	40000	96%
<i>Prime+Probe(V)</i>	1250000	45000	96%

We have noticed that, when monitoring more than one line at a time, each table provides different amount of information. This fact can be observed in Figure 6.8 where we represent a color map for each byte of the key candidates array for that byte. Dark colors (the darkest one in each horizontal line) represent less discarded values (correct key values) while light colors represent most discarded values. With a uniform distribution of the input plaintexts, the minimum score is more remarkable as we increase the number of encryptions. If we look into the figure, we see that some horizontal lines are darker than others. The ones with approximately the same darkness, correspond with bytes associated with the same table. We retrieve a different number of useful samples for each table. This is due to the prefetcher and the uniform access pattern that seems to trigger it. This means that more encryptions are required to get the full key.

We have been able to completely recover the full AES 256 key using the EBD attack. If we compare the number of traces required to obtain the 128 bit key and the number of traces required to obtain the 256 bit key, AES-256 happens to be only 3 times harder to attack with the non-accessed approach. The different probabilities of non-accessing a line explain the results: 2.69% for AES-256 and 7.5% for AES-128, that is in theory 2,8x more encryptions are needed to get the same information. In practice, we have measured 5.29%, 3.87%, 4.55% and 4.31% of cache misses within the samples for AES-128 and each of the lines respectively, and 1.52%, 0.93%, 1.31% and 1.02% for AES-256 for Flush+Reload.

The obtained results allow us to discuss the results in [175]. They state that although larger key lengths translate into an exponential increase in the complexity of a brute force approach this can not be applied to side-channel attacks. Also using the two last rounds of the encryption algorithm (a version that uses a different table for computing the last round of the encryption), they

Figure 6.8: Color map of discarding scores distribution for the 32 bytes of the key retrieved with the Flush+Reload technique. Correct key values are the darkest ones.

arrive at the conclusion that AES-256 only provides an increase in complexity of 6 to 7 compared to cache-based attacks on AES-128. We, in contrast, demonstrate that this factor is equal to 3.

In the case of the EBD attack, we have not found similar results or experimental setups to compare with. Indeed, the only practical attack against AES-256 was performed by Lapid et al [176]. Their target was the AES-256 and AES-256/GCM implementation used by the Keymaster trustlet in the ARM TrustZone. They used a GPU to accelerate the analysis of the samples. They required 40 minutes sample collection plus 30 more minutes of analysis to retrieve the key. However both their scenario and architecture are different than ours, although they also targeted a T-Table implementation. All the remaining studied attacks against AES 256 can be described as eminently theoretical.

On the literature, Aciğmez et al [177] stated that their attack can be applied to the second round of the encryption, but they do not provide experimental results. In 2009, Biryukov et al [178] presented a distinguisher and related key attack on AES-256 with 2^{131} data complexity. Later on, they slightly improved their results in [179], achieving 2^{119} data complexity. In [180] they describe an attack using two related keys to recover the 256-bit key, but their target cipher was the 9-round version of AES-256 and uses chosen-plaintexts. Another attack [181] runs in a related-subkey scenario with four related keys targeting the 13-round AES-256, it has 2^{76} time and data complexity. Bogdanov et al. [182] presented a technique of cryptanalysis with bicliques, also leading to a non-practical complexity recovering the key. Kim [183] explained also theoretically how to apply differential fault

6.2. Improving traditional attacks against AES

analysis to reduce the required number of faults in AES-192 and AES-256.

As stated before, each retrieved sample which carries information about non-access to the T-Tables reduces the key search space. This means that if the attack is interrupted for some reason before it finishes and the attacker decides to apply brute force to retrieve the key, the number of keys he has to check is much smaller. In Figure 6.9 we represent the base 2 logarithm of the key search space vs the total number of samples retrieved, that is the number of bits that remain unknown after each performed encryption. Note that in the case of the encryption with the 256 bit key we also begin with 128 unknown bits, as explained before we get the full key in two round using the same data and in each round we recover 128 bits. Thus, the figure represents the mean number of unknown bits per round.

Figure 6.9: Key bits that remain unknown after each encryption for the Flush+Reload technique

To put this data in context, we have also measured the time the two processes (victim and attacker) need to interact until we are able to obtain the full key. This time is referred to the measurement collection step, since the information processing is performed offline. Table 6.8 shows these mean times.

Table 6.8: Mean times required for gathering the data of Figure 6.9

	Attack duration
AES-128	43 ms
AES-128 (VM)	558 ms
AES-256	135 ms
AES-256 (VM)	1732 ms

Assuming the number of collected samples varies linearly with the time, these results show that if we want to detect such attacks before they succeed it is

6. Attacks

necessary to act fast. Detection times have to be in the order of milliseconds. Chiappetta et al. [67] let the victim run for 100ms, which means the non-access approach would succeed before they detect the attack. Even detection times of around 10ms, will imply that around half of the key will be leaked. In CloudRadar [66] detection times are approximately 5ms, then the hypervisor will need to migrate the machine. CacheShield [69] triggers an alarm after 4.48 ms, and will have to trigger later a countermeasure. It is realistic to assume that even with any of these countermeasures enabled, victim and attacker can interact for at least 8 to 10ms. This means part of the key will be leaked anyway, whereas there will be no leakage with the access approach. Note that we are assuming that the victim is encrypting random plaintext. If the attacker can make the victim encrypt chosen plaintext in such a way the number of non-accessed lines increases, the non-access approach would succeed much faster and trigger almost no cache misses, rendering previous countermeasures ineffective.

We have demonstrated that the proposed non-access approach retrieves the AES key using fewer samples than the traditional access approach in Intel processors that mount an inclusive cache. However, the AMD and ARM processors [50, 63, 64] that are vulnerable to cache attacks can also benefit from this attack. We believe it is possible to achieve a similar reduction in the number of samples required to succeed in these processors. Besides, it was also recently demonstrated that non-inclusive caches are also vulnerable to cache attacks [184]. As a result, the proposed approach is feasible for non-inclusive caches. As long as the attacker is able to evict the victim's data from the cache, he will be able to use the non-access approach to improve the efficiency of the attack against AES.

The T-Table implementation has been replaced by the S-Box implementation in software. In this case, the encryption is performed using a unique table holding 256 byte values. An attacker can also use the proposed non-accessed approach to target this implementation. If he is able to flush a line in the last round of the encryption, and given that the 16 access to the table per round represent a non-accessing probability of 1% (the attack is feasible), each non-accessed line would allow us to discard 64 values out of 256 possible of each byte of the key. Note that the presented attack against the T-table implementation discards 16 values of 256 possible for 4 bytes of the key. Thus, the estimated leakage of the theoretical attack to the last round of the new implementation is about 75 times higher than the presented attack to the T-table implementation. However, as we demonstrate in a later section, the main issue to attack the S-Box implementation is precisely flushing the data from the memory during the execution of the last round.

6.3 RELOAD+REFRESH

One of our main motivation when we reversed engineered the replacement policies implemented by current Intel processors was to check if we were able to bypass current cache-attack detection techniques. Since the implemented policy is completely deterministic, we were able to design an attack capable of extracting information from the victim causing negligible amount of misses and varying slightly its behavior. We have called this technique RELOAD+REFRESH, because we place data that the victim is likely to use in the cache, and use the ages of the elements to retrieve the desired information. Next, we need to REFRESH the cache, that is, manipulating the blocks in the set so the get the same ages values.

In order to use the same data that the victim may use, this approach relies on shared memory. If any kind of sharing mechanism is implemented, an attacker knowing the eviction policy can place some data that the victim is likely to use in the cache (the *target*) and in the desired position among the set. Since the position of the blocks and their ages (which in turn depend on the sequence of memory access) determine the exact eviction candidate, the attacker can force the *target* to be the eviction candidate. If the victim uses the *target* it will be no longer the eviction candidate, because its age decreases with the access. The attacker can force a miss and check afterwards if the *target* is still in the cache. If it is, the attacker has retrieved the desired information and the victim has retrieved the data from the cache without any cache misses (no attack trace). This is the main idea of the RELOAD+REFRESH attack.

OSs implement mechanisms such as Kernel Same-page Merging (KSM) in Linux [18] that improve memory utilization by merging multiple copies of identical memory pages into one. This feature was originally designed for virtual environments where multiple VMs are likely to place the same data in memory, and was later included in the OSs. Although most cloud providers have disabled it, it is still enabled in multiple OSs. When enabled, the attacker needs some reverse engineering to retrieve the address he wants to monitor and he also needs to find an eviction set that maps to the same set as this address. That is, the same previous step than is required for a FLUSH+Something attack. Section 4.1.3 shows how to construct the eviction sets in order to find the one which creates a “conflict” with the target address. In order to determine which set collides with the target, we follow the procedure in algorithm 3 replacing the conflicting set candidates with the target address.

6. Attacks

- 1) The attacker fills the set with the Target address and all the elements of the eviction set except one

Target	2	ev_0	2	ev_1	2	ev_2	2	ev_{w-2}	2
--------	---	--------	---	--------	---	--------	---	------------	---

- 2) The attacker waits. The next state of the cache depends on whether the victim accesses (a) or not (b) the target address

a)

Target	1	ev_0	2	ev_1	2	ev_2	2	ev_{w-2}	2
--------	---	--------	---	--------	---	--------	---	------------	---

b)

Target	2	ev_0	2	ev_1	2	ev_2	2	ev_{w-2}	2
--------	---	--------	---	--------	---	--------	---	------------	---

- 3) The attacker forces a miss by reading ev_{w-1} . The evicted element depends on whether the victim had accessed (a) or not (b) the Target.

a)

Target	2	ev_{w-1}	2	ev_1	3	ev_2	3	ev_{w-2}	3
--------	---	------------	---	--------	---	--------	---	------------	---

b)

ev_{w-1}	2	ev_0	3	ev_1	3	ev_2	3	ev_{w-2}	3
------------	---	--------	---	--------	---	--------	---	------------	---

- 4) The attacker now reloads the Target address, it will be placed in different positions depending on the previous accesses.

a)

Target	1	ev_{w-1}	2	ev_1	3	ev_2	3	ev_{w-2}	3
--------	---	------------	---	--------	---	--------	---	------------	---

b)

ev_{w-1}	2	Target	2	ev_1	3	ev_2	3	ev_{w-2}	3
------------	---	--------	---	--------	---	--------	---	------------	---

- 5) The attacker has to revert the changes. To ensure the Target gets age 2 and its placed in the “first” position, ev_{w-1} has to be flushed, as it has to be the Target, then the Target is reloaded and finally ev_0 is loaded

a)

Target	2	ev_0	2	ev_1	3	ev_2	3	ev_{w-2}	3
--------	---	--------	---	--------	---	--------	---	------------	---

b)

Target	2	ev_0	2	ev_1	3	ev_2	3	ev_{w-2}	3
--------	---	--------	---	--------	---	--------	---	------------	---

- 6) Accessing the remaining elements of the set will allow the attacker to get the cache as it was on the beginning.

a)

Target	2	ev_0	2	ev_1	2	ev_2	2	ev_{w-2}	2
--------	---	--------	---	--------	---	--------	---	------------	---

b)

Target	2	ev_0	2	ev_1	2	ev_2	2	ev_{w-2}	2
--------	---	--------	---	--------	---	--------	---	------------	---

Figure 6.10: Sequence of possible cache set states during the attack for the mode 1 or the newest generations. Note that all elements in the set start with age 2 and end with the same age.

The figure 6.10 depicts the stages of the attack and the possible “states” of the cache set. The attacker first inserts into the cache the target address and then all the elements in the eviction set except one, which will be used to force an eviction. By the time the attacker has finished filling the cache with data, the target address will be in level 3 cache. The number of ways in low level caches is lower than the number of ways in the L3 cache, and since the L3 cache is inclusive it will remove the target address from the low level caches when loading the last elements of the eviction set. Even if the victim and the attacker are located in the same core, an access of the victim to the target address will update its age,

so the attacker would be able to retrieve this information.

The data is placed in such a way that the target becomes the eviction candidate. The attacker then waits for the victim to access the target. If it does the eviction candidate changes and is now the element inserted in the second place. If it does not, the eviction candidate is still the target address. The attacker reads then the remaining element of the eviction set (ev_{w-1}), forcing this way a conflict in the cache set, and the eviction of the candidate. As a consequence, when reading (RELOAD) the target address again, the attacker will know if the victim has used the data (low reload time) or not (high reload time). The state of the cache has to be reverted to the initial one, so all the elements get the same age again (REFRESH). The element ev_{w-1} is forced out of the cache, so it could be used to create a new conflict on the next iteration.

Algorithm 10 Reload function

Input: Eviction_set, Target_address

Output: Reload time

```

function RELOAD(Target_address,eviction_set)
    "rdtsc";
    "mfence";
    read(eviction_set[w - 1]);                                ▷ Forces a miss
    "mfence";
    flush(eviction_set[w - 1]);
    "mfence";
    read(Target_address);
    flush(Target_address);
    "mfence";
    read(Target_address);                                    ▷ Reload on first position
    "mfence";
    "rdtsc";
    read(eviction_set[0]);
    return time_reload;
  
```

When the cache policy is working in mode 2, each element is inserted with age 3. In this case, steps 1 to 5 are equivalent. However step 6 changes depending on whether the victim is allocated in the same core as the attacker or not. When not, the other elements have age 3 and the target is the eviction candidate, so there is no need to refresh the data for the attack. On the other hand, when they are on the same core, the attacker needs to remove the target from the low level cache by refreshing the other elements in the cache. Moreover, the attack could be performed using the low level caches.

The mode 2 policy enables a detectable fast cross core cache attack that does not require shared memory. Once the cache set is filled with the attacker's data, the eviction candidate is now the first element inserted by the attacker. If the

6. Attacks

victim uses the expected data, the eviction candidate will be replaced. Even if the victim uses the data multiple times, its age will not change, since it will be fetched from the low level caches. Then, after forcing a miss, the attacker only has to access the first element (eviction candidate) to check whether the victim has or has not accessed the target data. Note that with this access the attacker replaces the victim's data (because it became the eviction candidate when loaded) so it is equivalent to the REFRESH. If, on the contrary, the victim does not use the data, the attacker's data will still be in the cache. The attacker will then flush and reload this data to ensure it gets age 3 again.

Algorithm 11 Refresh function

Input: Eviction_set

Output: Refresh time

```
function REFRESH(Eviction_set)
    volatile unsigned int time;
    asm __volatile__(
        " mfence \n"
        " rdtsc \n"
        " movl %%eax, %%esi \n"
        " movq 8(%1), %%rdi \n"           ▷ Eviction_set[1]
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"           ▷ Eviction_set[w-2]
        " mfence \n"
        " rdtsc \n"
        " subl %%esi, %%eax \n"           ▷ Time value on %eax
    );
    return time_refresh;
```

Algorithms 10 and 11 summarize the steps of the RELOAD+REFRESH attack when the insertion age is two (newest Intel generations or mode 1 in oldest generations). The cache set is filled with the target address plus $W - 1$ elements of the eviction set during initialization. Then, the attacker waits for the victim to run the code. Later, he performs the RELOAD and REFRESH steps. The RELOAD step gives information about the victim access and the REFRESH step gets the set ready to retrieve information from the victim. Note that we measure the whole REFRESH function, to get extra information about possible misses

that might have occurred. When initializing the set, we first fill the set, then flush the whole set and finally reload the data again to ensure the insertion order.

In the RELOAD function it is not necessary to flush the *Target_address* unless it has not been used by the victim. The same assumption is true for the conflicting address or the element $w - 1$ of the eviction set, which would have to be flushed only in that situation. However, to avoid *if* conditions in the code, we have chosen to implement the RELOAD function this way. Low reload times mean the data was used by the victim, high reload times mean it was not.

The REFRESH function presented is meant for a 12 way set. Since the target and the first element of the eviction set have been loaded in the RELOAD step, the REFRESH function only has to access the remaining 10 elements of the set. To avoid out of order execution and ensure the order, the elements of the eviction set have to be provided as a linked list (one element contains the address of the following one). Additionally the *refresh time* can be used to detect if any other process is also using that set.

6.3.1 Covert channel

In order to study the resolution of the proposed technique and to characterize it, as well as to compare it with previous approaches (FLUSH+RELOAD and PRIME+PROBE) we construct a covert channel between two processes (referred as *sender* and *receiver*) in a similar way as previous works have done [?, 71, 65].

The *sender* transmits a 1 by accessing a memory location from a shared library and a 0 by not-accessing it. Once the memory location is accessed, he waits for a fixed time and reads that data again. The receiver monitors the cache utilization using each of the aforementioned techniques and determines whether a 1 or a 0 was transmitted. That is, whether the victim has used the data or not. The *sender* and the *receiver* are not synchronized.

In the case of the PRIME+PROBE attack, we do not use the zig-zag pattern that was intended to avoid changes in the replacement policy [71]. We access the eviction set of size W always in the same order, and the elements are accessed as a linked list. Starting in a known state of the cache, this means that at most we need 2 probes to evict the data from the cache, if the access to the target happens in the middle of a probing stage. Gruss et al. [65] achieve eviction rates over 99%, but their approach involves to empirically determine the number of congruent elements that have to be accessed to ensure eviction and how many times.

In each of the experiments executed, the *sender* reads 50000 times the same memory location for each of the fixed waiting times. We consider as true positives

6. Attacks

when the *sender* access a piece of data and the *receiver* detects that access and similarly true negatives are non-access that are classified as 0. In some situations, the processor appears to be performing other tasks that do not allow the retrieval of information. Since we do not classify these samples, we do not consider them for evaluation. The experiments were performed in the i5-7600K machine (Table 4.1).

The results of these experiments in terms of the F-Score for each idle time (transmission of 0) are presented in the Table 6.9. Note that when the waiting time between samples is low, both PRIME+PROBE and RELOAD+REFRESH are not able to distinguish between 1 and 0. PRIME+PROBE presents slightly better resolution in our test system. Note that in this case, we do consider as false positives the samples that represent a 1 when the sender was idle. Otherwise, our F-Score would be 1 and still no 0 would be detected.

6.3.2 Noise tolerance

The proposed attack relies on the order in which the elements are inserted into the cache set to both avoid misses on the victim side and to learn information about the data that has been accessed. If other processes are running and using data that maps to the same cache slice (introducing noise), the efficiency of the attack can be lessened and also some detection mechanisms can be triggered.

As mentioned before, the refresh step can reveal such situations. Then, the attacker can slightly change the approach. Assuming that only one address is being used by the noise generating process, the attacker can easily handle noise, avoid detection and still gain information about the victim. The trick to deal with noise is placing the target on a different place within the set (the second place in this example). In case somebody else uses any data mapping to that set, the replaced data belongs to the attacker; specifically it is the data placed in first place in the set. When the attacker forces a miss, the eviction candidate will be either the target address (if the victim did not use it) or the element inserted in third place (the victim did use the target data). The attacker can gain information about the victim by reloading the target address and he must begin by refreshing the third element of the eviction set and finish with the first one which will evict the “noise” from the cache, so the age of all the blocks is 2 again.

Compared with traditional attacks such as PRIME+PROBE or FLUSH+RELOAD, our proposal is more sophisticated and involves extra steps in order to retrieve the same information. RELOAD+REFRESH includes an access to one address not located in the working set (the conflicting address) plus some other access focused in checking if the data has been replaced and in re-allocating the data

Table 6.9: F-Score for the different attacks when the *sender* accesses the data at different and fixed intervals (ns)

Times ι	50000	10000	1000	750	500	250
R+R	0.988	0.975	0.925	0.684	-	-
F+R	0.999	0.995	0.996	0.991	0.989	0.981
P+P	0.934	0.911	0.873	0.716	0.548	-

in the desired places within the cache set. As a consequence, it induces some misses on the attacker side and the corresponding delays. However, its resolution is enough to retrieve cryptographic keys as we show in the following section.

6.3.3 Results

Even when RELOAD+REFRESH has lower resolution than other attacks, it can be used to retrieve secret keys of cryptographic implementations. We demonstrate it and replicate two published attacks: one against the T-Table implementation of AES and one against the square and multiply exponentiation implementation included in RSA. Although both implementations have been replaced by new ones, we use them for comparison. All the experiments presented in this section have been performed in the intel i5-7600K machine (Table 4.1).

6.3.3.1 Attacking AES

The T-Table implementation used to be a popular software implementation of AES. While still available, this implementation is not the default option when compiling the Openssl library due to its susceptibility to micro architectural attacks. We use it here as an example of the ability of our proposal to extract sensitive information from a victim process.

Our scenario is similar to the one described by Irazoqui et al. [31], which was later replicated by us improving the amount of required samples to retrieve the key (section 6.2). Both approaches focused in retrieving information about the last round of the AES encryption process, in which the ciphertext is obtained by performing one XOR operation between an element contained in the tables and the secret key. As the content of the tables is publicly available from the source code, they obtained the secret key xoring it with the ciphertext.

Besides performing the attack against the AES T-Table implementation (Openssl 1.0.1f) using the RELOAD+REFRESH (R+R) technique, we have performed the same attack using the FLUSH+RELOAD (F+R) and PRIME+PROBE (P+P) techniques, to provide a fair comparison regarding to the number of traces required to obtain the key. In order to retrieve the whole key, the attacker has to

6. Attacks

monitor at least one line of each T-Table. The attacker can monitor from one up to four lines at a time. For this comparison we monitor one table at a time.

Table 6.10 shows the results for each of the approaches. In this scenario the attacker performs one operation, then the victim performs the encryption, and finally the attacker retrieves the information about the victim. That is, victim and attacker do not interfere with each other while doing the different operations. To obtain the key we use cache misses [?], so false positives are measured misses when the victim used the data in the Table. We repeat each experiment until we have recovered the key 10000 times.

Table 6.10: Mean number of samples required to retrieve each four byte group of the whole AES key when monitoring one line per encryption and their F-Score.

Attack	R+R	F+R	P+P
Samples	3800	3500	3900
F-Score	0.98	0.99	0.97

Measurement of LLC misses

RELOAD+REFRESH is able to retrieve an AES key with a negligible impact on the victim process. We compare the number of cache misses the victim suffers per encryption performed, for all the attacks and for normal executions. We use the PAPI software interface [29] to read the counters referred to the victim. PAPI allows us to insert one instruction just before and another one just after the encryption process to read the L3 cache misses counter, which is mainly the information used for cache attack detection [67, 66, 69, 68]. As a result, figure 6.11 shows the distribution of the number of misses the victim sees for each attack and for the normal execution of the encryption.

As figure 6.11 shows, our attack can not be distinguished from the normal performance of the AES encryption process by measuring the number of misses. When performing the PRIME+PROBE attack against AES, we avoid the zig-zag pattern and access the data in the same order every time. Previous experiments that we conducted showed that the eviction rate we achieved with that pattern was below 80%.

Additionally, we use the *rdtsc* instruction to measure the time it takes to complete each encryption and show the results in figure 6.12. The differences observed in figure 6.12 between the normal encryption and the RELOAD+REFRESH approach are not significant if we compare them with the other attacks. The mean encryption time when there is no attack is 595 cycles, whereas it increases up to 623 cycles when attacked with the RELOAD+REFRESH technique. This time difference exists because, when suffering the RELOAD+REFRESH attack, the

Figure 6.11: Distribution of the number of misses induced in the victim process by the different attacks, and with no attack. Each includes 1 million samples

Figure 6.12: Distribution of the encryption times in different situations. Each distribution includes 1 million samples.

victim has to load the data from the L3 cache instead of loading it from the L1 or L2 caches.

6.3.3.2 Attacking RSA

In this section we focus in the *square and multiply* exponentiation algorithm (section 2.7.2) [33] as Yarom et al. did [14]. As in the case of AES, this implementation has been replaced by constant-time implementations with no key dependant memory accesses.

Square and multiply computes $x = b^e \bmod m$ as a sequence of Square and Multiply operations (followed by a Modulo Reduce) that depend on the bits of the exponent e . If the bit happens to be a 1, then the Square-Multiply-Reduce sequence of operations is executed. On the other hand, if the bit is a 0, only the Square-Reduce operations are executed. As a consequence, retrieving the sequence of operations executed means recovering the exponent; that is, the key.

As a difference with the attack against AES, we monitor instructions instead of data. Additionally, an attack against RSA needs to have enough time resolution to correctly retrieve the sequence of operations. As we did with AES, we performed

6. Attacks

Figure 6.13: Example of a retrieved trace referred to an execution of a RSA decryption. The blue bars represent the real execution of squares (points equal to 800) and multiplies (700). The yellow line represents the information retrieved, low reload times mean detection of the multiply execution.

the attack using our stealthy technique as well as the FLUSH+RELOAD and PRIME+PROBE techniques.

In this work we use the *libgcrpyt* version 1.5.0, which includes the aforementioned square and multiply implementation. The key length in our experiments is 2048 bits and we collected information for 1000 decryptions per attack. When attacking RSA it is possible to monitor all the functions implied in the exponentiation or just one. When monitoring all the instructions, the attacker is able to reconstruct the sequence of observations. When monitoring only one instruction, the attacker has to use the differences of times between occurrences of the monitored event to retrieve the key. We only monitor the multiply operation.

Figure 6.13 shows an example of part of a retrieved trace using the presented RELOAD+REFRESH approach. The trace corresponding to the real sequence of squares and multiplies is represented as blue bars with different values: 800 means a square was executed and 700 it was a multiply. Since the *timestamp* is collected after each exponentiation operation has finished, and the *timestamp* of the attack samples after the reload operation, it may seem that there is some misalignment between traces.

The results of our experiments are summarized in table 6.11. As we did in the case of the covert channel analysis, we do not classify as false positive or false negative the samples that are lost, that is, not collected in time. This situation happens for about 1-2% of the samples. Since we try to detect multiply operations, false positives refer to the situation in which a multiply was detected but not executed. The accuracy is given as the number of correctly classified samples (True positives+True negatives) divided by the number of collected samples during the RSA decryption.

Table 6.11: Percentage of samples correctly retrieved and false positives generated by each approach when attacking RSA.

Attack	R+R	F+R	P+P
Accuracy	96.1%	98.6 %	95.4%
F-Score	0.952	0.99	0.945

Measurement of LLC misses

We have monitored the number of cache misses detected when executing a complete RSA decryption. When trying different keys, we have observed that the distributions change not only depending on the attack, but on the secret key. For this reason, the total amount of misses per encryption can not be used to detect ongoing attacks.

We have also monitored the victim LLC misses periodically, with a sampling rate of $100 \mu s$. We have collected samples for 1000 complete RSA decryptions in each of the scenarios. The results obtained in this case show a varying number of misses during the initialization steps. During this initialization, considering exclusively the number of misses caused in the different scenarios, is not possible to distinguish between attacks and the normal operation. Later on, the number of misses gets stable and tends to zero during the normal operation. Similarly, this trend can be observed during the RELOAD+REFRESH attacks. On the contrary, both FLUSH+RELOAD and PRIME+PROBE cause a noticeable amount of misses. The concrete mean values of the misses are presented in Table 6.12.

Since detection mechanisms such as CacheShield [69], define a region in which some misses are tolerated to avoid false positives, and only cache misses are considered, our attack will not trigger an alarm. Figure 6.14 shows the section of the decryption process in which the number of misses has become stable.

Figure 6.14: Detail of a trace of misses measured each $100 \mu s$ for each of the approaches.

6. Attacks

Since the RELOAD+REFRESH approach (as well as the other attacks) are not synchronized with the decryption operation, there are situations in which both victim and attacker can try to access the target data simultaneously. If the victim tries to execute the multiply operation when the attacker is flushing and reloading the mentioned line, the victim may get a miss. Therefore, a few misses can be observed in figure 6.14 for our approach.

6.3.4 Detection evaluation

RELOAD+REFRESH causes a negligible amount of LLC misses on the victim process. Thus, existing detection techniques would fail to detect the attack unless adapted. With the aim of quantifying the effect that our proposal has on the victim, we have periodically monitored different counters when executing the attacks against AES and RSA.

Again, we have used PAPI to collect information from different counters. The sampling rate was set to 100 μ s. Considering that not all the counters can be read in parallel, we have repeated the experiments multiple times. We have merged the results when the intervals were in a range defined by its mean value $\pm$ a 10% of this mean. In the particular case of RSA, the samples that refer to the beginning of the execution have been removed (we focus on the stable part). Finally, we have randomly selected 10000 samples per algorithm and attack to conduct the analysis.

The results of the analysis are summarized in Table 6.12. Note that L2 cache misses report the same value as L3 accesses, and similarly L1 misses are L2 accesses, so only one of the values is included in the Table.

Table 6.12: Mean and variance of the different counters collected during the execution of the attacks against AES and RSA. The results were obtained using 10000 samples collected each 100 μ s for each scenario.

	Cycles	L3 misses	L3 accesses	L3 reads	L2 instruction misses	L2 accesses
AES Normal	152000 $\pm$ 750	0 $\pm$ 0.1	714 $\pm$ 62	697 $\pm$ 63	31 $\pm$ 7	2050 $\pm$ 74
AES R+R	148000$\pm$634	0$\pm$0.1	713$\pm$60	702$\pm$61	212$\pm$15	2035$\pm$114
AES F+R	184000 $\pm$ 3000	10 $\pm$ 2	676 $\pm$ 65	676 $\pm$ 66	186 $\pm$ 12	2000 $\pm$ 92
AES P+P	158000 $\pm$ 1200	19 $\pm$ 3	452 $\pm$ 140	451 $\pm$ 139	209 $\pm$ 18	1810 $\pm$ 95
RSA Normal	336000 $\pm$ 11000	14 $\pm$ 26	100 $\pm$ 206	99 $\pm$ 204	7 $\pm$ 12	139 $\pm$ 316
RSA R+R	374000$\pm$38000	26$\pm$14	233$\pm$114	231$\pm$113	60$\pm$23	308$\pm$163
RSA F+R	364000 $\pm$ 50000	127 $\pm$ 38	200 $\pm$ 110	199 $\pm$ 108	97 $\pm$ 54	285 $\pm$ 240
RSA P+P	363000 $\pm$ 55000	112 $\pm$ 61	177 $\pm$ 159	174 $\pm$ 156	78 $\pm$ 83	211 $\pm$ 222

As it can be inferred from the Table 6.12, a single counter referring to L3 misses or accesses can not be used to distinguish attacks and the normal operation for both algorithms. In the particular case of L3 accesses, it could be used for RSA but not for AES. However, the L2 instruction misses counter, could distinguish between attacks and non-attacks for both algorithms. Note that if the sampling

rate of the attack is reduced, the number of L2 misses would similarly be reduced. As a solution, this value could be normalized respect to the total number of instructions executed. RELOAD+REFRESH changes the performance of the system in an observable way in the low level caches.

Our results demonstrate that counters referring to the LLC are not enough to detect RELOAD+REFRESH. Then, the assumption of previous detection mechanisms [66, 67, 69] that LLC misses or accesses reveal the attacks does not hold for RELOAD+REFRESH. Existing detection systems should be adapted or re-trained to include additional information about low level cache events.

6.3.5 Discussion of the results

The absence of randomness in the replacement algorithm makes it possible to accurately determine which of the elements located in a cache set will be evicted in case of conflict. Also, the accurate timers included in Intel processors altogether with the *cflush* instruction allow to trace access to the different caches and to force the cache lines to have the desired ages, we exploit these facts to run undetectable attacks.

RELOAD+REFRESH is just one way to exploit the eviction policy assuming some kind of memory sharing mechanism enabled. In the case that the victim and the attacker do not share memory, our proposal can be adapted so the attacker can track his own data. The attacker has to prepare the data in the set, in such a way that once the victim places his data in the cache, the eviction candidate is one of the elements the attacker controls. The target address will only be evicted if not used by the victim, who will see a cache miss the next time that tries to use it. If there is an encryption process going on, the most likely situation is that the victim uses the data. As a result, the number of misses will be limited.

The knowledge of the eviction policy, enables the usage of a different access pattern to gain the information about the victim and to ensure its data is really evicted from the cache, reducing the amount of false positives. Thus, PRIME+PROBE attacks, EVICT+RELOAD attacks or any attack requiring to evict some data from the cache can benefit from our results. Moreover, we believe that the PROBE step can, in some cases, be reduced to just one access to the eviction candidate.

We have not explored the capabilities of the RELOAD+REFRESH attack in the situation where the cache eviction policy is the one we name mode 2. We leave that exploration as well as the design of new access patterns to improve PRIME+PROBE attacks for future work.

6.4 Attack against AES S-Box implementations

In the previous sections we have already showed that the T-Table implementation of AES is vulnerable to cache attacks. For that reason, it was replaced in OpenSSL by a different implementation that tries to use the AES-NI instructions if possible. If not, the default software implementation is known as the S-Box implementation. Even when the S-Box implementation is not the default option in openSSL if the function is used from the command line, it is used if some code that uses the library calls the function *AES_encrypt()*.

We will later discuss further details about this implementation but, as for now, we just want to mention that the S-Box implementation included in OpenSSL prefetches the data in the S-Box before executing each round. This prefetching stage is intended as a countermeasure against cache attacks. If the attacker flushes a line holding data from the S-Box before or during the execution of any intermediate round, and waits until the encryption has been performed, he will only observe cache hits. Consequently, he would not be able to distinguish whether those access are due to the actual utilization of the line or to the prefetch stage.

When analyzing the S-Box implementation we noticed that the last round is still vulnerable. In theory if an attacker is able to remove the data from the cache at the precise instant after the prefetch stage has finished, he will be able to retrieve the desired information. That was one of the motivations of the subsequent experiments, we wanted to evaluate the capability of an attacker to perform cross core cache attacks that require removing the data from the cache at precise instants of time, using just the information retrieved from the cache. That is, we assume no synchronization between the victim and the attacker.

6.4.1 AES S-Box implementation in OpenSSL

AES [25] is a commonly used symmetric block cipher that operates with data in blocks of 16-bytes. As we have already stated, it consists of different operations (*AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns*) that are repeated each round. As a difference with the T-Table implementation, the **S-Box** implementation does not merge two operations into one. The **S-Box** is the table that holds the data for the *SubBytes* operation.

Consequently, this implementation only uses one table with 256 byte values (the **S-Box**). The table is used 16 times each round, one per input byte. Considering a cache line size of 64 bytes, such table uses 4 cache lines. If we compute the probability of not accessing the cache line, it is 0, as shown in equation. 6.11.

$$Pr[no\ access\ S_{Box}] = \left(1 - \frac{64}{256}\right)^{10 \cdot 16} = 0 \quad (6.11)$$

Additionally, this implementation includes a prefetch stage. Since the 256 bytes of the **S-Box** table map to 4 different cache lines, the encryption process only has to read 4 values to ensure the whole **S-Box** table is loaded in the cache memory. Even when the S-Box implementation performs memory access dependent on the secret key, the data will be always in the cache. As a result, traditional side channel attacks are not supposed to extract information from this implementation.

Figure 6.15: Histograms of the Probe times and the encryption times using the S-Box.

Indeed, if we compare the time it takes to perform an encryption with the time it takes to perform a *Probe* in one set, it is clear that it is not possible to get enough resolution with the PRIME+PROBE attack between round. The histograms of both processes are depicted in Figure 6.15.

Thus, one of the main challenges of your proposal is to design a technique or procedure to force one element of the cache out at desired times. Note that we focus in the last round of the encryption, however the encryption process is not time-constant. As a consequence even if we interrupt the victim whenever we want, we may or may not interrupt at the proper time. If we interrupt before the last prefetch, we will only observe *cache hits* in contrast if we force the miss after the encryption had ended, we will see *cache misses*. Whereas if we hit during the last round the observations will depend on the actual instant when the eviction happened.

Figure 6.16 represents the probability of observing a *cache miss* as a function of the number of operations that have been performed prior to the instant of interruption. For example, if we cause an interruption prior to the instruction 1,

Figure 6.16: Probability of not accessing a line during the last round of the encryption depending on the number of instructions of the last round that have been actually executed.

(just after the prefetch) we will observe around 1% of cache misses. However if we manage to evict the data just after the last operation (16), such probability of occurrence will increase up to 75%.

When analyzing the code of the implementation of OpenSSL we noticed that there are 4 different tables containing the data of the S-Box. We noticed that once our server process is running, it will always use the same S-Box. However, when we launched other processes that also had to use the S-Box, they use one of the four tables which is not the one being used by our server.

This particular S-Box implementation was attacked by Moghimi et al. [109]. Their considered scenario was different from ours. Their intention with their tool *Cachezoom* was to demonstrate that SGX is not only vulnerable to cache attacks, but also allows to obtain even more fine grain information about the code running inside the enclave via side-channels. Indeed, they used the L1 cache as source of information and were able to retrieve various samples per round and to distinguish the prefetching stages from the normal operations of the round.

6.4.2 Scenario

We assume no synchronization between the victim and the attacker. That is, the attacker does not know when the victim performs an encryption and has to retrieve that information. In order to do that the attacking process only monitors the cache memory, particularly, one of the lines of the S-Box. In our considered scenario, which is depicted in Figure 6.17, we can find the following agents:

- Server: Encrypts a block of data whenever it gets a request from any client.

Figure 6.17: Diagram of the considered scenario for the attack against the S-Box implementation of AES. The attacker and the victim share the hardware

- Client: Sends requests at random times, each around $100\mu\text{s}$ plus a random time. This process tries to emulate the normal behavior of a real network.
- Attacker: Monitors the cache to detect whenever there is an encryption going on, then waits for a fixed time intended to achieve an eviction in the last round of the encryption. This eviction will be caused either by flushing or by accessing one memory location as we will explain in the following sections.

Note that this is a simplified version of a real server process, but it is enough to create a realistic scenario for the attacks and to explain how the attack works. From the attacker perspective there are now two main challenges he must overcome. On the one hand he needs to detect when the victim is performing an AES execution. On the other hand, once he knows that the encryption has started, he needs to design a technique that allows him to evict the target data from the memory at the desired time.

6.4.3 Encryption detection

The prefetching stage of the targeted OpenSSL AES software implementation involves that just before starting the encryption, the S-Box will be loaded into the cache. As a consequence, monitoring one of the lines of the S-Box reveals the execution of an encryption. We observe one of the lines and whenever we detect it has been accessed, we assume the encryption has started. In order to do that, we follow two different approaches: we use the FLUSH+RELOAD [14] and the PRIME+ABORT techniques. The PRIME+ABORT technique [81] is more

6. Attacks

accurate cause we have observed that it gets the abort as soon as the conflict happens. However, not all processors include TSX, and for this reason we also include the evaluation using FLUSH+RELOAD .

Flush+Reload based detection

Figure 6.18: Percentage of correctly detected encryptions and valid encryptions for an attack as a function of the waiting time measured as the limit of a count. That is, we use a counting thread as clock

The FLUSH+RELOAD technique requires the victim and the attacker to share the library. This is not a problem, for example, if they are running Ubuntu 16.04 or Centos 7.1. We have observed that both distributions are shipped with a compiled version of *libcrypto* (OpenSSL) that will be shared among all the users.

As stated before, the S-Box implementation of AES included in the OpenSSL library, has four different S-Boxes. As a previous step, we had to determine which of them is used by the server. In our case is the third table, and it seems to keep the same through executions. We suspect it is the first choice for OpenSSL if there is no other process using this library. To retrieve this information, an attacker can just perform an unmodified version of the FLUSH+RELOAD or even PRIME+PROBE attacks.

For this evaluation, we require collaboration from the server. By collaboration we mean that we have modified the server so it gives us information about the precise time instants at which encryption begins and ends. We later use this information to evaluate how many of the encryptions have been detected and if so, how many are detected before the encryption had ended. Our goal is to find the most accurate approach to detect the encryption process.

The attacker flushes the data from the cache waits for an arbitrary time and

6.4. Attack against AES S-Box implementations

reloads. We have evaluated two different ways of waiting: increasing the value of a variable up to a limit or using the `rdtsc` instruction to set waiting times, and constantly checking if we have waited for the desired time. Each approach yields to different results, but they are similar.

Figure 6.18 shows the results in terms of correctly detected encryptions and valid encryptions (the encryption has detected before the victim had actually finished it) as a function of the waiting time. The waiting time refers to the limit up to which a variable is increased: from 0 to the limit. As it can be derived from the figure, limits around 20 obtain the best results in terms of valid encryptions. As a consequence, an attacker should select that limit to maximize his chances of success.

Algorithm 12 Generic attack algorithm against OpenSSL AES S-Box implementation

Input: `Address(S-Box(0)),Eviction_set` ▷ Address of the S-Box
Output: $\bar{X}_0, \bar{S}^{N_r+1}$ ▷ Information about the access and ciphertext

```

1: for  $t = 0$  to number_of_encryptions do
2:   Start_Transaction();
3:   if Successfully Started then
4:     Prime_cache_set();
5:   else ▷ Abort handler, detects the access
6:     time_interruption = timestamp() + WAIT_TIME;
7:     While (timestamp() ≤ time_interruption) {};
8:     Evict from cache (S-Box(0)); ▷ Data is removed WAIT_TIME
      cycles after detection
9:     Wait until encryption ends
10:    Infer victim access to((S-Box(0))) ▷ RECOVERING
11:    if hasAccessed((S-Box(0))) then
12:       $\bar{X}_0[t] = 1$ ; ▷ The attacker decides if the victim has used the data.
13:    else
14:       $\bar{X}_0[t] = 0$ ;
15:    end if
16:  end if
17: end for
18: return  $\bar{X}_0, \bar{S}^{N_r+1}$ ;

```

Prime+Abort based detection

As we did in the previous case, we use the data collected from the server to evaluate the accuracy of the detection. This scenario requires the aforementioned eviction set, that is, we first have to create the eviction sets and find which one maps with the S-Box. Then we access all the elements in the set inside a

6. Attacks

transactional region as if we were to perform a PRIME+ABORT attack.

At the moment the victim access the data of the S-Box we get an abort. In the abort handler, we have measured the instant of time at which we get the abort. If we compare this time with the time at which the encryption begins, the difference between them is almost constant. Indeed, the value of this difference is about 380 cycles in our system (Intel Core i5-7600K). Note that this time includes the time of the first cache miss. That is, we do not get the abort until the data of the S-Box has been effectively load into the cache.

This behavior clearly benefits our attacks, since it almost does not introduce noise or variations in our measurements. This way, we are able to properly detect 97% of the encryptions and they are all valid. The "undetected" encryptions are likely to be the ones when the victim has already loaded its data into the cache, and we are still placing our data in the region.

6.4.4 Hitting the last round

Due to the prefetching stage of this implementation, the ability of the attacker to retrieve the secret key is determined by his ability to evict the data from the cache during the last round of the encryption. Again, we consider two different possibilities, depending on the existence of shared memory or not. In this section, we explain the particular techniques used to achieve the eviction. Since the Prime+Abort detection obtained better results, from now on, we always perform the detection using that approach.

In both cases, the procedure we will follow to evict the data during the last round of the encryption process is summarized in the algorithm 12. Note that both approaches will differ in the value of **WAIT_TIME**, the way the target data is evicted from the cache and the way the information about the actual access to the data is inferred. As a result the attacker gets the ciphertext and the information about the access to the S-Box. Next, we explain the particularities of each approach.

6.4.4.1 Prime+Abort+Wait+Flush+Reload

Since this approach *flushes* the victim's data from the cache, it assumes shared memory. Once we have detected that the victim is performing an encryption, in the abort handler we actively wait for a **WAIT_TIME** until we *flush* the data from the memory. Then we have to wait again for a while until the encryption has finished. After that, we **RELOAD** the data and use the reload time to determine if the data in the S-Box has been accessed or not. This part behaves as a normal **FLUSH+RELOAD** attack.

Note that an attacker does not know in advance the value of **WAIT_TIME**. In order to retrieve this information automatically we run a test that tries different values of the parameter and computes the number of cache hits and misses for a fixed number of measurements. We use then the percentages between misses and number of encryptions performed to decide the value of **WAIT_TIME**. Ideally, we expect to observe a distribution similar to the defined in Figure 6.16 if we hit the last round of the encryption. In practice, the encryption does not execute in constant time and there are features such as pipelining and out of order execution that are out of our control. For this reason, we select a threshold that allows around an 8% of misses.

We have tried two different approaches. We have used directly the threshold (**WAIT_TIME**) obtained during the duration of all the experiments. And we have used an adaptive threshold. That is, we run the attack for W encryptions, where W is a length of a window we will use to evaluate the relation between cache misses and W , and decide whether we need to reduce the value of the **WAIT_TIME** or increase it. If we measure too many misses we are probably evicting the data either at the end of the round or either when the encryption has ended. In both cases, we will reduce the value of **WAIT_TIME**. In our experiments we used windows of about 10000 samples. This approach helps to improve our results in case the threshold has not been properly set.

6.4.4.2 Prime+Abort+Wait+Access+Access

If there is no shared memory, it is not possible to prime the whole set to achieve the desired eviction. We demonstrate that if we can manipulate the cache prior to the detection of the encryption (in the `Prime_cache_set()` step of the algorithm 12), one access is enough to evict the data from the memory. To this end, we use the knowledge of the replacement policy that we previously have reverse engineered (section 4.1). The first thing we have checked is that the manipulations to the elements of the cache we do inside a transaction, are maintained after the abort. Then we proceed to explain the particular details of our approach.

Transient is transient

Transient execution attacks [6, 7] exploit the fact that some instructions are executed by the processor even if the program never gets to see the results of these operations. We are going to manipulate the data into the transactional region in order to later retrieve the information in the abort handler. We want to ensure that what happens inside the transactional region also remains in the architecture.

6. Attacks

One of the simplest ways to check this is flushing some data out of the cache before executing the transactional code. Then, in the transactional region, reload that data back into the cache. Next, we wait in an endless loop doing nothing within the transaction, until an abort happens. The abort handler checks if the data is in the cache or not. We have repeated the experiment around 10000 times, and the conclusion is that the data that has been loaded into the cache during a transactional operation, remains in the cache after such operation has finished.

Replacement policy and accurate eviction of data

To understand this approach there are two things we need to bear in mind. The first one is that only access to the last level cache update the values of the ages of the elements in the LLC. The second one is that the LLC in our machine is inclusive. This means that if the victim loads the S-Box into the cache, and uses it multiple times during the encryption, the age of the cache line holding the S-Box will keep the same. As a result, if it is the oldest element in the set it will be evicted in case of conflict.

The general idea of the attack is depicted in Figure 6.19. As a difference with the previous approach the attacker needs to set the ages of the elements in the set to the ones shown in the second stage of the figure. This means that the `Prime_cache_set()`; stage in the algorithm 12 may involve two steps. Note that since we use that step for detection all the elements in the eviction set must be accessed.

Once the encryption begins, the S-Box data will be placed into the cache. There will be no elements with age 3 due to the previous probe phase, so the ages of all the elements in the set will increase after the cache miss. The first element in the set, where we want the S-Box data to be placed, should be older than the remaining ones by one unit. This way, since the insertion age is 2 all the elements in the set will get age 2 right after the S-Box data enters the cache. And the line holding the S-Box data becomes the eviction candidate.

Then the procedure is similar as it was in the previous case. We wait for the desired time and then by accessing A (which is out of the cache because it was replaced with the S-Box data) we ensure that the data of the S-Box is removed from the cache. We wait until the encryption ends, and finally retrieve the information about the actual access to the S-Box by accessing B. Since B is the first element with age 3, in case of conflict it will be replaced with the S-Box data. Note that after the recovery step we need to change the ages again.

The procedure to get the proper value of **WAIT_TIME** is similar as in the previous case. The only difference is that this waiting time is shorter. While the `clflush` is immediate and evicts the data from the cache as soon as it is called, the

6.4. Attack against AES S-Box implementations

access is not so fast. Indeed the data is only evicted from the cache when A has been retrieved from the main memory.

Figure 6.19: General diagram of the process required to evict the data from the cache with just one access.

An observation on the replacement policy of low level caches

When trying to force all the elements in the set to get the ages depicted in the stage 2 of Figure 6.19, we accessed the elements from B to H in an ordered way as a linked list (to avoid pipeling effects as much as possible). To ensure that the data had really been retrieved from the LLC, we measured the times, and found out that some of them were retrieved from the L1 cache. The replacement policy of L1 and L2 caches was responsible for this effect, so we must take it into account.

We performed further experiments to understand how it works, and found out a replacement policy that seemed to match our observations. We provide further details in section 4.2. Note that if the set size of L1 was 8 and the size of the set in L3 is at least two times bigger, we would not be required to consider the replacement policy in L1, since all the data would have already been evicted by the time the reading process of the data in L3 had finished.

In fact, our processor LLC has 12 ways, whereas the L1 and the L2 caches have 8 ways and 4 ways respectively. Just after the 12 elements of the set have been placed in the cache, 4 of them are only present in the LLC, and the L1 cache will have suffered 4 misses. The replacement policy defines which elements are replaced. However, if the number of ways of the LLC is 16, as in a different processor in which we have performed some tests (Intel i7-6700K), the L1 cache

6. Attacks

will have suffered 8 misses. This means it is likely that the 8 elements that were first accessed, only reside in the LLC. We observed that in the first case the block we call B is in the L1 cache whereas it is not in the second case.

We have already described the replacement policy in low level caches (tree based Pseudo-LRU). Since we want to decrease the age of all the elements, we need to access B when it is in the LLC. The replacement policy in the L1 switches the values or the nodes whenever an element in the cache is either accessed or evicted and replaced. Since we know the order in which the elements were inserted in the cache, we know which elements are still in the cache and finally we can define a linked list of addresses to access the 11 elements in our 12 way cache that ensures all of them are in the LLC when accessed. As mentioned before, this is how we are able to reduce their age.

Once we have build that list of 11 addresses, we have checked that all the elements in such list are indeed retrieved from the LLC. We have measured the time that it takes to retrieve each element and, as expected, the reading times are consistent with LLC access times.

6.4.5 Results

In this section we present the results in terms of number of encryptions that the attacker needs to monitor to retrieve the whole key. All the experiments were performed in the machine described in table 6.13. The version of openssl used is the one included in our CentOS system (OpenSSL 1.0.2k-fips).

Table 6.13: Experimental platform details.

Processor	Cores	Frequency	OS	LLC slices	LLC size	LLC ways
Intel core i5-7600K	4	3.8 GHz	CentOS 7.6	8	6 MB	12 ways

In order to retrieve the secret key in all of scenarios we use the information retrieved during the attack (algorithm 12). We use the *non-access* approach presented in section 6.2. If we determine that an element has not been accessed it means none of the operations in the last round has used it. As a consequence we xor each byte of the ciphertext with the 64 values of the S-box hold in the cache line ($k_i = C_i \oplus \text{S-Box}[0 \text{ to } 63]$).

Both approaches, the one that assumes shared memory and the one that does not, are able to successfully retrieve the secret AES key. Even when we do not hit exactly the encryption process after the last prefetch round, if we hit it a few times is enough. Indeed, as our results demonstrate, it is more likely to evict the data in the middle of the last round than just at the beginning. As a result some bits are recovered faster than others as it can be observed in Figure 6.20. The number of samples required to retrieve the secret key slightly varies between

Figure 6.20: Key candidates for each of the bytes of the key of the S-Box AES implementation of OpenSSL retrieved using the Prime+Abort + Count + Flush+Reload approach.

executions and depends on the threshold selected, even if we use the adaptive approach. In both cases, the minimum number of samples required to retrieve the whole key is about 500000.

As it can be observed in Figure 6.20, half of the key has been completely leaked after 100000 samples. Around 12 of the 16 bytes are already known with 200000 samples. Retrieving the last 4 bytes of the key is the hardest part, it takes 30000 more samples, which gives an idea about the difficulty of evicting the data in the last round.

Figure 6.21 is complementary to Figure 6.20. It represents the number of guesses that an attacker would require to retrieve the key using brute force and the information about the actual access of the victim. Note that both plots represent the same experiment. Different experiments have yielded to similar results but, as we have stated before depend on the actual threshold defined by **WAIT_TIME**. Besides, we have noticed that the location of the S-Box in the cache is also important, because there are some sets that are noisier than others.

Note that this attack does not intend to be stealthy, so if the victim is implementing any detection mechanism based on times or on performance counters, he will for sure detect it is under attack. indeed this attack is possible because

6. Attacks

Figure 6.21: Key candidate search space for the 128 bits of the key of the S-Box AES implementation of OpenSSL retrieved using the Prime+Abort+ Count+ Flush+Reload approach.

the last round of the encryption does not include one last prefetch. Adding such instructions at the end of the encryption process will prevent the leakage, at least through the LLC and with no further assumptions as the SGX scenario [109].

As we have demonstrated, it is possible to achieve precise eviction at the desired times by combining different techniques and using the detailed knowledge of the eviction policy we have achieved. We have demonstrated that even without deduplication or any kind of shared memory, we are still able to achieve enough resolution.

Conclusions

In the last years processor designers have designed optimized architectures that try to make the most of the processor in terms of performance. As a result they have favored multicore architectures, and features such as concurrent and out-of-order execution, branch prediction and so on. They have trusted the logical isolation between processes or virtual machines to the OS or the hypervisor respectively. At the same time, they have disregarded the potential hardware threats due to the physical sharing of the hardware.

Side-channel attacks, use the observable effects that the execution of a process produces. Particularly, microarchitectural side-channel attacks use microarchitectural elements to retrieve information from a victim process. These attacks do not cause any damage in the hardware, not require any special privilege to succeed and are not identified as malicious by most anti-viruses. Although they exploit the hardware, they are triggered by software, even remotely. For example, if a user opens a browser and visits a malicious site. Among the multiple existing side-channels, in this work we mainly focus in the cache as a mean to obtain secrets from the processor.

These attacks keep gaining increasing attention from the research community due to their ability to extract private information from their victims in relatively short time. For instance, researchers have used microarchitectural side-channels to retrieve AES, RSA, ECDSA cryptographic keys among other private information exploiting the shared hardware. Cloud computing environments, where multiple virtual machines share the same hardware are not extent of this threat. Even the recently discovered vulnerabilities known as Spectre and Meltdown and all their variants rely on a microarchitectural element to gain information about a victim.

In spite of the great threat microarchitectual attacks, and particularly cache

7. Conclusions

attacks, pose in existing architectures, we have not observed any change in the hardware design. Besides, no Operating System or cloud provider is implementing any of the proposed countermeasures. We observe the necessity of protection that users of the existing hardware have, and aim to protect them from these attacks. Since we cannot modify the existing hardware nor the Operating System, we opted for the detection of the attacks while they are being executed.

As a difference with previous approaches, we focus on the user. We do not require collaboration from the OS, nor the hypervisor to detect the attacks. In order to develop practical countermeasures, there must be a trade-off between security and performance. Thus, our proposed countermeasures are effective and introduce low overhead to the system at the same time. In order to develop them, we have studied and evaluated how state of the art cache attacks influence and change the behavior of their victims. Using elements available in the processor such as timers or hardware performance counters, we have demonstrated that the effects caused by an attack in the victim process are noticeable, and can be used to detect such attacks.

We have analyzed existing cache attacks that target well-known encryption algorithms and we have improved the speed of the key recovery process up to 98% by changing the approach. Not only we speed up the attack but we also demonstrate a practical attack against AES256. These results greatly limit the amount of time allowed for the detection. Otherwise, the attacker succeeds. We have studied the effects that one process has in their neighbors in terms of performance degradation. By quantifying such degradation we have been able to infer the CPU load of the system in which is running the monitored process. That information can be used either as a covert channel, as a mean to check if two processes are sharing the same hardware or even to extract information about the state of the host.

In order to further understand how the cache works and, in turn, how cache-attacks work, we have reverse engineered the replacement policy implemented in modern Intel core processors. Indeed we have observed two different policies depending on the processors. To the best of our knowledge, we are the first ones to provide a comprehensive description of the policy up to the point that we are able to predict exactly which element of the cache set is going to be evicted. We first reveal the policy implemented in the shared last level cache, and we later extend the analysis to the lower level caches.

Further, we use the knowledge of the replacement policy to obtain information from a victim process without causing cache misses on it. We name the attack RELOAD+REFRESH and it is based on the manipulation of the ages of the memory blocks allocated in the cache. Prior to this attack demonstration, cache attacks had to remove the victim's data from the cache, generating an anomalous

rate of cache misses on the target process. Such information is frequently used for detection. Our attack renders detection mechanisms based on cache hits/misses ineffective. As a consequence, detection mechanisms have to be redesigned to protect against this attack.

Finally, we combine some ideas and advantages of the different attacks with the knowledge of the replacement policies, to design a mechanism that allows the attacker to interrupt the victim at the desired instant. We demonstrate that shared memory is not necessary to evict the data from the cache without accessing the whole eviction set. To this end just one access to a memory location is enough. We demonstrate the accuracy of our approach by retrieving the secret key of the S-Box based implementation of AES, that includes prefetching as a countermeasure against cache attacks.

7.1 Future work

Although many of the features or the elements included in modern processors have already been exploited, there are still many others that have not. We set as an example the particular case of Spectre and Meltdown. They opened a new area of research and triggered the disclosure of multiple vulnerabilities related to them. As of today, the security community and processor designers are not sure about how many of them remain undisclosed or about how to solve them in a way that does not greatly impact the performance. Fixing the bugs while keeping the performance of the processor as high as possible still requires some joint work of security researchers and microarchitecture designers.

Besides, every day new attacks are designed and exploited, and unknown and novel side channels are very likely to exist. For this reason, it is necessary to keep analyzing the possible leakages and the effects that one process has in his neighbors to provide stronger protection to the users.

Since one of the main contributions of this work is the retrieval of the replacement policy implemented in modern Intel Core processors, we plan to expand this work to ARM and AMD processors. A different idea is to build a hardware model of the cache, that features the implemented policy. This way, researchers will be able to deeply understand how the cache works, and to design and test new countermeasures that require minimum changes to the hardware. Hopefully, if they introduce low overhead or performance degradation chip manufacturers will be willing to implement them.

One future line, that we are currently developing, is related to energy efficiency. Providing that there is a monitoring infrastructure in both machines and datacenters, we believe that we can take advantage of such infrastructure to im-

7. Conclusions

prove performance, energy efficiency and security at the same time. Thus, we are working in the definition of an allocation policy pursuing such goals that we describe in the following section.

7.1.1 Allocation Policies and Active Monitoring

The VMM acts as a scheduler that allocates the different VM upon request based on the available resources and the requirements of each VM. Cloud services are provided under Service Level Conditions (SLA) conditions that must be respected. Optimizing resource utilization and energy consumption, minimizing this way the costs while satisfying the clients is one of the main goals of an allocation policy.

The workload of cloud data centers is highly variable and most of the times there is no prior information about which tasks a Virtual Machine is going to execute. Besides, previous works demonstrate that not all kinds of task interfere with the others in the same way [185]. For these reasons, defining optimal allocation strategies is a challenge in cloud environments [186]. There have been multiple proposals that aim to reduce energy consumption while maintaining the quality of the service. For example, Arroba et al. [187, 188] suggest different techniques as Dynamic Voltage and Frequency Scaling (DVFS) in order to reduce the energy consumption in data centers.

During the development of this thesis, we have observed that building expected behavioral models is a hard task without the collaboration of client VMs. Besides, we have demonstrated that the interaction between malicious agents and legacy process greatly affects their performance. The performance of each VM and its variations can be measured by the owner of the infrastructure. Taking into account the historical records of the performance of the VMs, plus the known behavioral patterns of different attacks, we propose to provide the VMM with a decision supporting system that considers energy saving and performance optimization as a baseline and improves the security of the data center at the same time.

We propose to implement a Trust and Reputation system that actively monitors the different VMs allocated in the host. This system can be seen as a supervisor that is able to detect certain kind of attacks and performance issues in the system and acts in consequence. With the information collected and provided by this system, the VMM can take advantage of cloud technologies as live migration of VM when detecting an attack, isolate malicious VMs or allocate in the same host different VM that cause negligible interference among them.

7.1.1.1 Considered threats

Most of the attacks that target the cloud infrastructure such as Rowhammer attacks, execute for several minutes. Thus, a monitoring system that intends to detect these attack does not have strict timing requirements. As we have previously demonstrated a monitoring system that retrieves information from the hardware at low rates, generates negligible overhead.

On the other hand, there are other attacks, cache attacks, for instance, that would require milliseconds to succeed. Monitoring the different VMs would incur in a huge overhead in the system. For these reason, we leave cache-attacks out of the scope of our general proposal. However, if a tenant is concerned about these attacks, the cloud provider can offer a specific surveying system that seeks for anomalies in the cache at a cost for such tenant.

In this work, we consider Rowhammer attacks [129, 102] that can lead to privilege escalation and compromise the infrastructure of the cloud provider, performance degradation attacks and other different and generic malware such as Ransomware attacks. This last group of attacks encrypt the victim's data and ask for a payment to the victim in order to recover its data back.

7.1.1.2 Trust and Reputation Systems

We propose to implement active monitors in the cloud system as Trust and Reputation Systems (TRSs) [189] that support the decision making process when allocating new VMs in the hosts. TRSs map traditional social concepts of trust and reputation with numeric values that will support the aforementioned decision making process. Trust is a subjective concept that can be defined as the belief of one entity that the outcome of other entities actions is going to be in a specific way. Reputation is defined as the collective idea that a group or entities within a system have about another entity based on a specific criterion. Trust and reputation systems have been used to improve security in wireless sensor networks, to detect thermal anomalies in data-centers, to allocate VMs with energy reduction proposals, etc.

One advantage of TRSs is that they can be defined to actualize their trust and reputation values at fixed time intervals and later change this time if needed. Thus, TRSs can be adjusted to not cause a performance degradation while still detecting anomalies. Several TRSs can also co-reside, which means that we can have a TRS in charge of security and another one in charge of minimizing energy consumption and use both when making a decision about allocating a VM or about the need of performing a migration.

Cloud providers continuously monitor the VMs, our proposal implies extend-

7. Conclusions

ing the applicability of such monitoring system. Having introduced these trust and reputation systems, we are going to explain briefly how to use them using the information retrieved from the monitoring subsystem.

In particular, we define Trust between the different VMs allocated in a host. If two virtual machines have never seen each other in advance, the trust they have in each other is rather neutral. The trust between them increases if they do not significantly alter each others performance. The actual values of the increments depend on the mutual interference.

Similarly, the host is in charge of assigning the Reputation to each VM. VMs that use high amount of resources, get smaller Reputation values to favor low energy consumption. The reputation of a VM also decreases if an anomalous behavior pattern or an attack is identified. If the trust between all the allocated VMs and a particular VM decrease simultaneously, the global reputation of that machine decreases accordingly and it is considered pervasive.

Both Trust and Reputation are actualized dynamically, so the VMs can similarly be reallocated dynamically. Besides, the scheduler can use the information about the current status of the system, the expected energy consumption and the trust between the different VMs in order to decide where each VM will be placed. Additionally, the system can be easily extended to consider more parameters adjusting the proper parameters of the TRS.

Bibliography

- [1] “Skylake (client) - microarchitectures - intel,” <https://en.wikichip.org/wiki/WikiChip>, 2015, last accessed on September 15, 2019.
- [2] I. W. Paper), “5-level paging and 5-level ept,” 2017. [Online]. Available: https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf
- [3] B. Gregg, “Linux performance.” [Online]. Available: <http://www.brendangregg.com/linuxperf.html>
- [4] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. Francillon, “Reverse engineering intel last-level cache complex addressing using performance counters,” in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, ser. RAID 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 48–65.
- [5] P. C. Kocher, “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *Advances in Cryptology - CRYPTO*, ser. LNCS, vol. 1109. Springer, 1996, pp. 104–113.
- [6] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *40th IEEE Symposium on Security and Privacy (S&P’19)*, 2019.
- [7] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Melt-down: Reading kernel memory from user space,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 973–990.
- [8] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-tenant side-channel attacks in paas clouds,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14. New York, NY, USA: ACM, 2014, pp. 990–1003.

BIBLIOGRAPHY

- [9] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds,” in *ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, 2009, pp. 199–212.
- [10] D. Page, “Theoretical use of cache memory as a cryptanalytic side-channel,” *IACR Cryptology ePrint Archive*, vol. 2002, p. 169, 2002. [Online]. Available: <http://eprint.iacr.org/2002/169>
- [11] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, “Cryptanalysis of DES implemented on computers with cache,” in *Cryptographic Hardware and Embedded Systems - CHES*, ser. LNCS, vol. 2279. Springer, 2003, pp. 62–76.
- [12] D. J. Bernstein, “Cache-timing attacks on aes,” Tech. Rep., 2005.
- [13] D. A. Osvik, A. Shamir, and E. Tromer, “Cache Attacks and Countermeasures: The Case of AES,” in *Topics in Cryptology – CT-RSA 2006: The Cryptographers’ Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.
- [14] Y. Yarom and K. Falkner, “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.
- [15] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.
- [16] W. Stallings, *Computer Organization and Architecture: Designing for Performance (10th Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2015.
- [17] T. W. Barr, A. L. Cox, and S. Rixner, “Translation caching: Skip, don’t walk (the page table),” *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 48–59, Jun. 2010.
- [18] A. Arcangeli, I. Eidus, and C. Wright, “Increasing memory density by using KSM,” in *OLS ’09: Proceedings of the Linux Symposium*, Jul. 2009, pp. 19–28.
- [19] “Kernel samepage merging,” http://kernelnewbies.org/Linux_2.6.32#head-d3f32e41df508090810388a57efce73f52660ccb/, 2015.

- [20] Intel, “Intel® 64 and ia-32 architectures optimization reference manual (section 2.1.1.2),” 2017, last accessed on September 15, 2019. [Online]. Available: <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf>
- [21] F. J. Corbato, “A paging experiment with the multics system,” p. 20, 07 1968.
- [22] I. Sun Microsystems, “Ultrasparc t2 supplement to the ultrasparc architecture 2007,” Draft D1.4.3, Sep 2007.
- [23] H. Wong, “Intel Ivy Bridge cache replacement policy,” jan 2013. [Online]. Available: <http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/>
- [24] P. Vila, B. Köpf, and J. F. Morales, “Theory and practice of finding eviction sets,” in *2019 2019 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2019, pp. 695–710.
- [25] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, ser. Information Security and Cryptography. Springer, 2002.
- [26] B. Mann, “New technologies for the arm a-profile architecture.” [Online]. Available: <https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/new-technologies-for-the-arm-a-profile-architecture>
- [27] V. Costan and S. Devadas, “Intel sgx explained,” *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.
- [28] Intel, “Intel® 64 and ia-32 architectures software developer’s manual: Volume 3,” 2017, <https://www.intel.es/content/www/es/es/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html>.
- [29] P. J. Mucci, S. Browne, C. Deane, and G. Ho, “Papi: A portable interface to hardware performance counters,” in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.
- [30] J. Levon, “Oprofile,” <http://oprofile.sourceforge.net>, 2003, last accessed on September 15, 2019.
- [31] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, “Wait a minute! A fast, cross-vm attack on AES,” in *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, 2014, pp. 299–319.

BIBLIOGRAPHY

- [32] C. Koç, “Analysis of sliding window techniques for exponentiation,” *Computers & Mathematics with Applications*, vol. 30, no. 10, pp. 17 – 24, 1995.
- [33] D. M. Gordon, “A survey of fast exponentiation methods,” *J. Algorithms*, vol. 27, no. 1, pp. 129–146, Apr. 1998.
- [34] F.-X. Standaert, “Introduction to side-channel attacks,” in *Secure Integrated Circuits and Systems*. Springer, 2010, pp. 27–42.
- [35] D. Brumley and D. Boneh, “Remote timing attacks are practical,” in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM’03. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1.
- [36] B. B. Brumley and N. Tuveri, “Remote timing attacks are still practical,” in *Proceedings of the 16th European Conference on Research in Computer Security*, ser. ESORICS’11, 2011, pp. 355–371.
- [37] J. Ferrigno and M. Hlavac, “When aes blinks: introducing optical side channel,” *IET Information Security*, vol. 2, no. 3, pp. 94–98, Sep. 2008.
- [38] L. Zhuang, F. Zhou, and J. D. Tygar, “Keyboard acoustic emanations revisited,” *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 3:1–3:26, Nov. 2009.
- [39] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, “Acoustic side-channel attacks on printers.” in *USENIX Security symposium*, 2010, pp. 307–322.
- [40] D. Genkin, A. Shamir, and E. Tromer, “Rsa key extraction via low-bandwidth acoustic cryptanalysis,” in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461.
- [41] E. Peeters, F.-X. Standaert, and J.-J. Quisquater, “Power and electromagnetic analysis: Improved model, consequences and comparisons,” *Integration*, vol. 40, no. 1, pp. 52 – 60, 2007, embedded Cryptographic Hardware.
- [42] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, “The em side—channel(s),” in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 29–45.
- [43] P. C. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO ’99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 388–397.

- [44] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, Apr 2011.
- [45] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, M. Joye and J.-J. Quisquater, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 16–29.
- [46] J.-S. Coron, D. Naccache, and P. Kocher, "Statistics and secret leakage," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 492–508, Aug. 2004.
- [47] R. Mayer-Sommer, "Smartly analyzing the simplicity and the power of simple power analysis on smartcards," in *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '00. London, UK, UK: Springer-Verlag, 2000, pp. 78–92.
- [48] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 13–28.
- [49] W. Schindler, K. Lemke, and C. Paar, "A stochastic model for differential side channel cryptanalysis," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, J. R. Rao and B. Sunar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 30–46.
- [50] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: ACM, 2016, pp. 353–364.
- [51] W. Hu, "Lattice scheduling and covert channels," in *IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1992, pp. 52–61.
- [52] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *Journal of Computer Security*, vol. 8, no. 2/3, pp. 141–158, 2000.
- [53] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Fine grain cross-vm attacks on xen and vmware are possible!" *IACR Cryptology ePrint Archive*, vol. 2014, p. 248, 2014. [Online]. Available: <http://eprint.iacr.org/2014/248>
- [54] C. Percival, "Cache missing for fun and profit," in *Proc. of BSDCan 2005*, 2005.

BIBLIOGRAPHY

- [55] D. Gullasch, E. Bangerter, and S. Krenn, “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice,” in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 490–505.
- [56] B. Gülmözoglu, M. S. Inci, G. I. Apecechea, T. Eisenbarth, and B. Sunar, “A faster and more realistic flush+reload attack on AES,” in *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, 2015, pp. 111–126.
- [57] N. Benger, J. Pol, N. P. Smart, and Y. Yarom, ““ooh aah... just a little bit”: A small amount of side channel can go a long way,” in *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2014 - Volume 8731*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 75–92.
- [58] Y. Yarom and N. Benger, “Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack,” *IACR Cryptology ePrint Archive*, vol. 2014, p. 140, 2014.
- [59] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, “Lucky 13 strikes back,” in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 85–96.
- [60] —, “Know thy neighbor: Crypto library detection in cloud,” *PoPETs*, vol. 2015, no. 1, pp. 25–40, 2015.
- [61] D. Gruss, R. Spreitzer, and S. Mangard, “Cache template attacks: Automating attacks on inclusive last-level caches,” in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 897–912.
- [62] X. Zhang, Y. Xiao, and Y. Zhang, “Return-oriented flush-reload side channels on arm and their implications for android devices,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 858–870.
- [63] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, “ARMageddon: Cache Attacks on Mobile Devices,” in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 549–564.

- [64] M. Green, L. R. Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, “Autolock: Why cache attacks on ARM are harder than you think,” *CoRR*, vol. abs/1703.09763, 2017. [Online]. Available: <http://arxiv.org/abs/1703.09763>
- [65] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+flush: A fast and stealthy cache attack,” in *13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.
- [66] T. Zhang, Y. Zhang, and R. B. Lee, *CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds*. Cham: Springer International Publishing, 2016, pp. 118–140.
- [67] M. Chiappetta, E. Savas, and C. Yilmaz, “Real time detection of cache-based side-channel attacks using hardware performance counters,” *Applied Soft Computing*, vol. 49, pp. 1162 – 1174, 2016.
- [68] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, “Spydetector: An approach for detecting side-channel attacks at runtime,” *International Journal of Information Security*, Jun 2018.
- [69] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, “Cacheshield: Detecting cache attacks through self-observation,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY ’18. New York, NY, USA: ACM, 2018, pp. 224–235.
- [70] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, “Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud,” *IACR Cryptology ePrint Archive*, Tech. Rep., 2015.
- [71] Fangfei Liu and Yuval Yarom and Qian Ge and Gernot Heiser and Ruby B. Lee, “Last level Cache Side Channel Attacks are Practical,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 605–622.
- [72] R. Hund, C. Willems, and T. Holz, “Practical timing side channel attacks against kernel space aslr,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 191–205.
- [73] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, “Mapping the intel last-level cache,” *IACR Cryptology ePrint Archive*, vol. 2015, p. 905, 2015.
- [74] G. Irazoqui, T. Eisenbarth, and B. Sunar, “Systematic reverse engineering of cache slice selection in intel processors,” in *2015 Euromicro Conference on Digital System Design (DSD)*, vol. 00, Aug. 2015, pp. 629–636.

- [75] M. Neve and J.-P. Seifert, “Advances on access-driven cache attacks on aes,” in *Proceedings of the 13th International Conference on Selected Areas in Cryptography*, ser. SAC’06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 147–162.
- [76] O. Aciğmez and W. Schindler, “A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and its Demonstration on OpenSSL,” in *Topics in Cryptology–CT-RSA 2008*. Springer, 2008, pp. 256–273.
- [77] C. Chen, T. Wang, Y. Kou, X. Chen, and X. Li, “Improvement of trace-driven i-cache timing attack on the {RSA} algorithm,” *Journal of Systems and Software*, vol. 86, no. 1, pp. 100 – 107, 2013.
- [78] B. B. Brumley and R. M. Hakala, “Cache-timing template attacks,” in *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ser. ASIACRYPT ’09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 667–684.
- [79] G. Irazoqui, T. Eisenbarth, and B. Sunar, “S\$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing and its Application to AES,” in *36th IEEE Symposium on Security and Privacy (S&P 2015)*, 2015, pp. 591–604.
- [80] M. S. İnci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, “Cache Attacks Enable Bulk Key Recovery on the Cloud,” in *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, B. Gierlichs and A. Y. Poschmann, Eds., 2016.
- [81] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime+abort: A timer-free high-precision l3 cache attack using intel TSX,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 51–67.
- [82] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, “Smotherspectre: exploiting speculative execution through port contention,” 2019.
- [83] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution,” in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018, see also technical report Foreshadow-NG [84].

- [84] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, “Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution,” *Technical report*, 2018, see also USENIX Security paper Foreshadow [83].
- [85] G. Maisuradze and C. Rossow, “Ret2spec: Speculative execution using return stack buffers,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: ACM, 2018, pp. 2109–2122.
- [86] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, “Netspectre: Read arbitrary memory over network,” 2018.
- [87] J. Wampler, I. Martiny, and E. Wustrow, “Exspectre: Hiding malware in speculative execution,” in *NDSS*, 2019.
- [88] M. Schwarz, S. Weiser, and D. Gruss, “Practical enclave malware with intel SGX,” *CoRR*, vol. abs/1902.03256, 2019. [Online]. Available: <http://arxiv.org/abs/1902.03256>
- [89] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. V. Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, “Fallout: Reading kernel writes from user space,” *ArXiv*, vol. abs/1905.12701, 2019.
- [90] M. Schwarz, C. Canella, L. Giner, and D. Gruss, “Store-to-leak forwarding: Leaking data on meltdown-resistant cpus,” *ArXiv*, vol. abs/1905.05725, 2019.
- [91] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss, “Zombieload: Cross-privilege-boundary data sampling,” *ArXiv*, vol. abs/1905.05726, 2019.
- [92] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “RIDL: Rogue in-flight data load,” in *S&P*, May 2019.
- [93] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gülmezoglu, T. Eisenbarth, and B. Sunar, “Spoiler: Speculative load hazards boost rowhammer and cache attacks,” *ArXiv*, vol. abs/1903.00446, 2019.
- [94] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtvyushkin, and D. Gruss, “A systematic evaluation of transient execution attacks and defenses,” *ArXiv*, vol. abs/1811.05441, 2018.

BIBLIOGRAPHY

- [95] M. Guarnieri, B. Koepf, J. F. Morales, J. Reineke, and A. Sánchez, “SPECTECTOR: Principled detection of speculative information flows,” in *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 2020.
- [96] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, “oo7: Low-overhead defense against spectre attacks via binary analysis,” *ArXiv*, vol. abs/1807.05843, 2018.
- [97] M. Schwarz, R. A. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, “Context: Leakage-free transient execution,” *ArXiv*, vol. abs/1905.09100, 2019.
- [98] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping bits in memory without accessing them: An experimental study of dram disturbance errors,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 361–372.
- [99] D. Gruss, C. Maurice, and S. Mangard, “Rowhammer.js: A remote software-induced fault attack in javascript,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 300–321.
- [100] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, “Drammer: Deterministic rowhammer attacks on mobile platforms,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: ACM, 2016, pp. 1675–1689.
- [101] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, “One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation,” in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 19–35.
- [102] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O’Connell, W. Schoechl, and Y. Yarom, “Another flip in the wall of rowhammer defenses,” pp. 245–261, May 2018.
- [103] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, “DRAMA: Exploiting DRAM addressing for cross-cpu attacks,” in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 565–581.

- [104] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, “Rambleed: Reading bits in memory without accessing them,” in *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.
- [105] M. S. İnci, G. Irazoqui, T. Eisenbarth, and B. Sunar, “Efficient, adversarial neighbor discovery using logical channels on microsoft azure,” in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ser. ACSAC ’16. New York, NY, USA: ACM, 2016, pp. 436–447.
- [106] T. Zhang, Y. Zhang, and R. B. Lee, “Dos attacks on your memory in cloud,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’17. New York, NY, USA: ACM, 2017, pp. 253–265.
- [107] M. S. Inci, T. Eisenbarth, and B. Sunar, “Hit by the bus: Qos degradation attack on android,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’17. New York, NY, USA: ACM, 2017, pp. 716–727.
- [108] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, “Rendered insecure: Gpu side channel attacks are practical,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: ACM, 2018, pp. 2139–2153.
- [109] A. Moghimi, G. Irazoqui, and T. Eisenbarth, “Cachezoom: How sgx amplifies the power of cache attacks,” in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, pp. 69–90.
- [110] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, “Port contention for fun and profit,” 2019.
- [111] O. Aciğmez, c. K. Koç, and J.-P. Seifert, “Predicting secret keys via branch prediction,” in *Proceedings of the 7th Cryptographers’ Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA’07. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 225–242.
- [112] —, “On the power of simple branch prediction analysis,” in *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS ’07. New York, NY, USA: ACM, 2007, pp. 312–320.
- [113] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Jump over aslr: Attacking branch predictors to bypass aslr,” in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 40:1–40:13.

BIBLIOGRAPHY

- [114] L. Uhsadel, A. Georges, and I. Verbauwhede, “Exploiting hardware performance counters,” in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, Aug 2008, pp. 59–67.
- [115] S. Bhattacharya and D. Mukhopadhyay, “Who watches the watchmen?: Utilizing performance monitors for compromising keys of rsa on intel platforms,” in *Cryptographic Hardware and Embedded Systems – CHES 2015*, T. Güneysu and H. Handschuh, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 248–266.
- [116] R. Martin, J. Demme, and S. Sethumadhavan, “Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 118–129.
- [117] D. Kohlbrenner and H. Shacham, “Trusted browsers for uncertain times,” in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 463–480.
- [118] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware,” *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, Apr 2018.
- [119] V. Costan, I. Lebedev, and S. Devadas, “Sanctum: Minimal hardware extensions for strong software isolation,” in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 857–874.
- [120] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA ’07. New York, NY, USA: ACM, 2007, pp. 494–505.
- [121] T. Kim, M. Peinado, and G. Mainar-Ruiz, “Stealthmem: System-level protection against cache-based side channel attacks in the cloud,” in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 189–204.
- [122] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, “Catalyst: Defeating last-level cache side channel attacks in cloud computing,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 406–418.

- [123] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, “Scattercache: Thwarting cache attacks via cache set randomization,” in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 675–692.
- [124] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, “On the feasibility of online malware detection with performance counters,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 559–570.
- [125] X. Wang and R. Karri, “Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters,” in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–7.
- [126] S. Bhattacharya and D. Mukhopadhyay, *Who Watches the Watchmen?: Utilizing Performance Monitors for Compromising Keys of RSA on Intel Platforms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 248–266.
- [127] B. Gregg, “The PMCs of EC2: Measuring IPC,” 2017, last accessed on September 15, 2019. [Online]. Available: <http://www.brendangregg.com/blog/2017-05-04/the-pmcs-of-ec2.html>
- [128] B. Gülmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar, “Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning,” *CoRR*, vol. abs/1907.03651, 2019. [Online]. Available: <http://arxiv.org/abs/1907.03651>
- [129] M. Seaborn, “Exploiting the dram rowhammer bug to gain kernel privileges,” March 2015, last accessed on September 15, 2019. [Online]. Available: <https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html>
- [130] A. Zankl, J. Heyszl, and G. Sigl, “Automated detection of instruction cache leaks in modular exponentiation software,” in *Smart Card Research and Advanced Applications: 15th International Conference, CARDIS 2016, Cannes, France, November 7–9, 2016, Revised Selected Papers*, K. Lemke-Rust and M. Tunstall, Eds. Cham: Springer International Publishing, 2017, pp. 228–244.
- [131] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying constant-time implementations,” in *25th USENIX Security Sympo-*

BIBLIOGRAPHY

- sium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 53–70.
- [132] A. Moghimi, T. Eisenbarth, and B. Sunar, “Memjam: A false dependency attack against constant-time crypto implementations in sgx,” in *Topics in Cryptology – CT-RSA 2018*, N. P. Smart, Ed. Cham: Springer International Publishing, 2018, pp. 21–44.
- [133] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, “Cacheaudit: A tool for the static analysis of cache side channels,” in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 431–446. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev>
- [134] G. Irazoqui, T. Eisenbarth, and B. Sunar, “Mascot: Preventing microarchitectural attacks before distribution,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY ’18. New York, NY, USA: ACM, 2018, pp. 377–388.
- [135] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, “Microwalk: A framework for finding side channels in binaries,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC ’18. New York, NY, USA: ACM, 2018, pp. 161–173.
- [136] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang, “Deepsec: A uniform platform for security analysis of deep learning model,” in *2019 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2019.
- [137] S. Jahagirdar, V. George, I. Sodhi, and R. Wells, “Power management of the third generation intel core micro architecture formerly codenamed ivy bridge,” in *2012 IEEE Hot Chips 24 Symposium (HCS)*, Aug 2012, pp. 1–49.
- [138] N. Megiddo and D. S. Modha, “Arc: A self-tuning, low overhead replacement cache,” in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, ser. FAST ’03. Berkeley, CA, USA: USENIX Association, 2003, pp. 115–130.
- [139] S. Bansal and D. S. Modha, “Car: Clock with adaptive replacement,” in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, ser. FAST ’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 187–200.

- [140] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, “A case for mlp-aware cache replacement,” in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 167–178.
- [141] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, “Adaptive insertion policies for high performance caching,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391.
- [142] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 60–71.
- [143] A. Abel and J. Reineke, “Measurement-based modeling of the cache replacement policy,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 65–74.
- [144] A. Abel and J. Reineke, “Reverse engineering of cache replacement policies in intel microprocessors and their evaluation,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 141–142.
- [145] Y. research, “Yahoo! Cloud System Benchmark (YCSB),” <https://github.com/brianfrankcooper/YCSB>, 2010, last accessed on September 15, 2019.
- [146] H. Liu, “A measurement study of server utilization in public clouds,” in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, Dec 2011, pp. 435–442.
- [147] P. Li, D. Gao, and M. K. Reiter, “Stopwatch: a cloud architecture for timing channel mitigation,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 2, p. 8, 2014.
- [148] V. Varadarajan, T. Ristenpart, and M. Swift, “Scheduler-based defenses against cross-vm side-channels,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 687–702.
- [149] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, “Homealone: Co-residency detection in the cloud via side-channel analysis,” in *2011 IEEE Symposium on Security and Privacy*, May 2011, pp. 313–328.

BIBLIOGRAPHY

- [150] M. B. Bahador, M. Abadi, and A. Tajoddin, “Hpcmalthunter: Behavioral malware detection using hardware performance counters and singular value decomposition,” in *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, Oct 2014, pp. 703–708.
- [151] A. Tang, S. Sethumadhavan, and S. Stolfo, “Unsupervised anomaly-based malware detection using hardware features,” in *Research in Attacks, Intrusions and Defenses*, ser. Lecture Notes in Computer Science, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Springer International Publishing, 2014, vol. 8688, pp. 109–129.
- [152] M. Payer, “Hexpads: A platform to detect “stealth” attacks,” in *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016. Proceedings*, J. Caballero, E. Bodden, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2016, pp. 138–154.
- [153] E. Page, “Continuous inspection schemes,” *Biometrika*, vol. 41, no. 1/2, pp. 100–115, 1954.
- [154] R. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun, “Thermal covert channels on multi-core platforms,” in *24th USENIX Security Symposium*. Washington, D.C.: USENIX Association, 2015, pp. 865–880.
- [155] B. Singh, D. Evtvyushkin, J. Elwell, R. Riley, and I. Cervesato, “On the detection of kernel-level rootkits using hardware performance counters,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 483–493.
- [156] S. Briongos, P. Malagón, J. L. Risco-Martín, and J. M. Moya, “Modeling side-channel cache attacks on aes,” in *Proceedings of the Summer Computer Simulation Conference*, ser. SCSC ’16. San Diego, CA, USA: Society for Computer Simulation International, 2016, pp. 37:1–37:8.
- [157] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The weka data mining software: An update,” *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.
- [158] K. Kira and L. A. Rendell, “The feature selection problem: Traditional methods and a new algorithm,” in *AAAI*, vol. 2, 1992, pp. 129–134.
- [159] Y. Yarom and N. Benger, “Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack,” *IACR Cryptology ePrint Archive*, vol. 2014, p. 140, 2014.

- [160] “Amazon cloudwatch,” Web Site, 2017, last accessed on September 15, 2019. [Online]. Available: <https://aws.amazon.com/cloudwatch/>
- [161] M. Basseville and I. V. Nikiforov, *Detection of Abrupt Changes: Theory and Application*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.
- [162] A. G. Tartakovsky, B. L. Rozovskii, R. B. Blazek, and H. Kim, “A novel approach to detection of intrusions in computer networks via adaptive sequential and batch-sequential change-point detection methods,” *IEEE Transactions on Signal Processing*, vol. 54, no. 9, pp. 3372–3382, Sept 2006.
- [163] D. McDonald, “A cusum procedure based on sequential ranks,” *Naval Research Logistics (NRL)*, vol. 37, no. 5, pp. 627–646, 1990.
- [164] A. G. Tartakovsky, B. L. Rozovskii, R. B. Blažek, and H. Kim, “Detection of intrusions in information systems by sequential change-point methods,” *Statistical Methodology*, vol. 3, no. 3, pp. 252 – 293, 2006.
- [165] V. M. D. Oca, D. R. Jeske, Q. Zhang, C. Rendon, and M. Marvasti, “A cusum change-point detection algorithm for non-stationary sequences with application to data network surveillance,” *Journal of Systems and Software*, vol. 83, no. 7, pp. 1288 – 1297, 2010, {SPLC} 2008.
- [166] R. Longbottom, “Randmem benchmark,” <http://www.roylongbottom.org.uk/>, 2016, last accessed on September 15, 2019.
- [167] T. El Gamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” in *Proceedings of CRYPTO 84 on Advances in Cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 10–18.
- [168] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, “Memory deduplication as a threat to the guest os,” in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC ’11. New York, NY, USA: ACM, 2011, pp. 1:1–1:6.
- [169] H. Liu, “A measurement study of server utilization in public clouds,” in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, Dec 2011, pp. 435–442.
- [170] L. J. Adams, G. A. Bello, and G. G. Dumancas, “Development and application of a genetic algorithm for variable optimization and predictive modeling of five-year mortality using questionnaire data,” *Bioinformatics and Biology Insights*, no. Suppl 3, pp. 31–41, 2015.

BIBLIOGRAPHY

- [171] R. Tibshirani, “Regression shrinkage and selection via the lasso,” *Journal of the Royal Statistical Society, Series B*, vol. 58, pp. 267–288, 1994.
- [172] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.
- [173] D. Carraway, “Lookbusy: a synthetic load generator,” <https://www.devin.com/lookbusy/>, last accessed on September 15, 2019.
- [174] MATLAB, *version (R2015a)*. Natick, Massachusetts: The MathWorks Inc., 2015.
- [175] M. Neve and K. Tiri, “On the complexity of side-channel attacks on AES-256 - methodology and quantitative results on cache attacks,” *IACR Cryptology ePrint Archive*, vol. 2007, p. 318, 2007. [Online]. Available: <http://eprint.iacr.org/2007/318>
- [176] B. Lapid and A. Wool, “Cache-attacks on the arm trustzone implementations of aes-256 and aes-256-gcm via gpu-based analysis,” in *Selected Areas in Cryptography – SAC 2018*, C. Cid and M. J. Jacobson Jr., Eds. Cham: Springer International Publishing, 2019, pp. 235–256.
- [177] O. Aciicmez, W. Schindler, and Ç. K. Koç, “Cache based remote timing attack on the AES,” in *Topics in Cryptology – CT-RSA*, ser. LNCS, vol. 4377. Springer, 2007, pp. 271–286.
- [178] A. Biryukov, D. Khovratovich, and I. Nikolić, “Distinguisher and related-key attack on the full aes-256,” in *Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO. Springer, 2009, pp. 231–249.
- [179] A. Biryukov and D. Khovratovich, “Related-key cryptanalysis of the full aes-192 and aes-256,” *Cryptology ePrint Archive*, Report 2009/317, 2009, <http://eprint.iacr.org/>.
- [180] A. Biryukov, O. Dunkelman, N. Keller, D. Khovratovich, and A. Shamir, “Key recovery attacks of practical complexity on AES-256 variants with up to 10 rounds,” in *Advances in Cryptology - EUROCRYPT*, ser. LNCS, vol. 6110. Springer, 2010, pp. 299–319.
- [181] A. Biryukov and D. Khovratovich, “Feasible attack on the 13-round aes-256,” 2010, khovratovich@gmail.com, alex.cryptan@gmail.com 14734 received 5 May 2010. [Online]. Available: <http://eprint.iacr.org/2010/257>
- [182] A. Bogdanov, D. Khovratovich, and C. Rechberger, “Biclique cryptanalysis of the full aes,” in *Advances in Cryptology - ASIACRYPT*, ser. LNCS, vol. 7073. Springer, 2011, pp. 344–371.

- [183] C. H. Kim, “Differential fault analysis of AES: toward reducing number of faults,” *Inf. Sci.*, vol. 199, pp. 43–57, 2012.
- [184] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, “Attack directories, not caches: Side channel attacks in a non-inclusive world,” in *2019 2019 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2019.
- [185] J. C. Salinas-Hilburg, M. Zapater, J. L. Risco-Martín, J. M. Moya, and J. L. Ayala, “Unsupervised power modeling of co-allocated workloads for energy efficiency in data centers,” in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, ser. DATE ’16. San Jose, CA, USA: EDA Consortium, 2016, pp. 1345–1350.
- [186] N. Hamdy, A. E. Aboutabl, N. ElHaggar, and M.-S. M. Mostafa, “Resource allocation strategies in cloud computing: Overview,” *International Journal of Computer Applications*, vol. 177, no. 4, pp. 18–22, Nov 2017.
- [187] P. Arroba, J. M. Moya, J. L. Ayala, and R. Buyya, “Dynamic voltage and frequency scaling-aware dynamic consolidation of virtual machines for energy efficient cloud data centers,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 10, p. e4067, 2017.
- [188] P. Arroba, J. L. Risco-Martín, J. M. Moya, and J. L. Ayala, “Heuristics and metaheuristics for dynamic management of computing and cooling energy in cloud data centers,” *Software: Practice and Experience*, vol. 48, no. 10, pp. 1775–1804, 2018.
- [189] A. Jøsang, “Trust and reputation systems,” in *Foundations of Security Analysis and Design IV, FOSAD 2006/2007 Tutorial Lectures*, 2007, pp. 209–245.