Towards Automated Analysis of Microarchitectural Attacks using Machine Learning

by

Berk Gulmezoglu

A Dissertation

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy in

Electrical and Computer Engineering
by

July 2020

APPROVED:

Professor Dmitry Ponomarev
Dissertation Committee
State University of New York at Binghamton

Professor Patrick R. Schaumont
Dissertation Committee
ECE Department

Professor Thomas Eisenbarth
Dissertation Co-Advisor
Universität zu Lübeck

Professor Berk Sunar
Dissertation Co-Advisor
ECE Department

Professor Donald R. Brown
Department Head
ECE Department

Abstract

Cloud computing has gained tremendous popularity among small and mid-size businesses, providing many companies to access cloud services without the need for investing in computer software and hardware. In order to offer the highest performance to clients, multi-core servers are mostly preferred by the cloud providers, which yields to sharing the same hardware resources among clients. Although it is expected that hardware virtualization is a sufficient protection against potential attackers from co-located users, microarchitectural attacks still pose a significant threat in the cloud due to the shared hardware resources. Moreover, similar attack techniques are applicable in both personal computers and mobile phones when a benign-looking malicious application is installed in the system. Today, microarchitectural attacks are more important threat than ever, since the capabilities of the attacks have been extended tremendously.

In order to recover confidential information from a third-party by implementing a microarchitectural attack, thousands or millions of side-channel measurements are collected. Since the side-channel analysis requires engineering expertise to extract the information such as handling mis-alignment and extracting the secret bits one by one, it takes huge amount of time to process millions of side-channel data samples. Thanks to recent advances in Machine Learning, the complex tasks can be handled efficiently with the high computation power of GPUs. The linear and non-linear problem solving capabilities of Machine Learning algorithms are integrated with giant matrix multiplications, the success rate improves drastically. However, the

appropriate application of Machine Learning techniques on side-channel measurements is still an ongoing research area, which can provide huge time and performance gain in terms of leakage extraction.

The goal of this dissertation is to propose Machine Learning based approaches to processing of side-channel measurements in different platforms. First, we introduce a targeted co-location technique by using cryptographic libraries on public clouds. Then, a full RSA key recovery attack based on last-level cache is demonstrated on Amazon EC2 public cloud. Next, we implement a Machine Learning assisted cache attack on a public cloud as well as showing that ping requests can be used to identify the co-located VMs. Furthermore, we show that privacy of personal computer and mobile phone users can be violated by third-party applications through microarchitectural attacks. For this purpose, we detect the visited websites, launched applications and watched trailers, as well as, comparing the performance of several Machine Learning techniques. Finally, we propose a Recurrent Neural Network based unsupervised detection mechanism for microarchitectural attacks. We achieve a low false alarm rate and performance overhead by combining the sequence learning capabilities of RNNs and computation power of GPUs.

Acknowledgments

The studies in this dissertation are funded by the National Science Foundation grants CNS-1618837, CNS-1318919, and CNS-1314770. I would like to thank the National Science Foundation for their support.

First, I would like to thank my advisor Prof. Berk Sunar for his support, advice and guidance that helped me to find the most suitable projects that I can work on. Especially, his guidance on Machine Learning topics helped me think about challenging problems the security community faces. I hope that our collaboration will continue and I will always remember his advice in my future career.

I also would like to thank my co-advisor Prof. Thomas Eisenbarth for his support and guidance. He always paid attention to details to improve my projects. He spent his valuable time to explain to me the cryptographic functions, architectural details and so on. I also hope that our collaboration will continue on many other projects.

I would like to thank my dissertation committee members Professor Berk Sunar, Professor Thomas Eisenbarth, Professor Dmitry Ponamorev, and Professor Patrick Schaumont. I am grateful for their guidance, feedback and their invaluable time spent for the preparation of this dissertation.

I have been fortunate to collaborate with awesome colleagues in the Vernam Lab. I would like to thank Mehmet Sinan Inci for helping me on both research and exploring the Worcester area in my first year. My sincere thanks also go to Yarkin Doroz

for his help with the immigration process. I also thank Koksal Mus for being there whenever I need help. I also would like to thank my dear friends Aria Shahverdi, Okan Seker, Gizem Cetin, Micheal Moukarzel, Wei Dai, Marc Green, Gorka Irazoqui, Daniel Moghimi, Saad Islam, Koray Yurtseven and Caner Tol for both their social and academic conversations. Finally, I am really grateful to Andreas Zankl for both his help during my Germany visit, as well as, his unique perspective on our projects.

I would also like to thank my parents for their support through my education since I started school. I always felt their confidence in me, especially at times when I was stressful. I will never forget how much time they spent with me to understand every single concept in my curriculum all the way to high school. Thanks to their guidance, I understood the importance of higher education for being successful in every area of my life.

Last but not least, I would like to thank my wife, Sumeyra Gok, for going through this journey with me. We faced many challenges in our career and reached many milestones together. I feel lucky to have her in my life and I am looking forward to spending the rest of my life with her.

Contents

1 Introduction **1**

1.1 Cryptography . . . 3

1.2 Cloud Security . . . 4

1.3 Microarchitectural Attacks . . . 5

1.4 Machine Learning and Deep Learning . . . 6

1.5 Problem Statement . . . 8

1.6 Thesis . . . 8

1.7 Contributions . . . 9

1.7.1 The publications resulted in this dissertation . . . 11

2 Background **12**

2.1 Computer Architecture . . . 13

2.1.1 Memory Hierarchy . . . 13

2.1.2 CPU Cache . . . 14

2.1.3 Cache Attacks . . . 15

2.2 Hardware Performance Events (HPEs) . . . 18

2.2.1 Profiling with Perf . . . 19

2.3 Machine Learning Techniques . . . 21

3 Related Work **26**

3.1 Co-location Detection . . . 26

3.2 Microarchitectural Attacks . . . 28

3.3 Other Microarchitectural Attacks . . . 30

3.4 Website Fingerprinting . . . 30

3.5 Machine Learning and Side-Channel Attacks . . . 31

3.6 Defense Mechanisms against Attacks . . . 32

4 Co-location Detection **34**

4.1 Motivation . . . 34

4.2 **Software Profiling on LLC** . . . 36

4.3 Results . . . 37

4.4 Outcome . . . 40

5 Microarchitectural Attacks in the Cloud **41**

5.1 Faster Flush+Reload Attack . . . 41

5.1.1 Motivation . . . 41

5.1.2 A single cache line attack on AES . . . 44

5.1.3 Distinguishers for the AES Attack . . . 47

5.1.4 Attack Scenarios . . . 50

5.1.5 Experiment Setup . . . 52

5.1.6 Results . . . 53

5.1.7 Outcome . . . 57

5.2 Prime+Probe Attack on Amazon Cloud . . . 58

5.2.1 Motivation . . . 58

5.2.2 Cross-VM RSA Key Recovery . . . 61

5.2.3 Leakage Analysis Method . . . 64

5.2.4 Outcome 68

6 Machine Learning based Application Detection in the Cloud 69

6.1 Motivation 69

6.2 Methodology 73

6.2.1 Extracting Feature Vectors from Applications on Cache 73

6.2.2 Extracting feature vectors from L1 cache 76

6.2.3 Extracting feature vectors from LLC 79

6.2.4 Targeted co-location by ping detection on the cloud 82

6.3 Application Detection Results 84

6.3.1 Experiment Setup 84

6.3.2 Application Detection in Native Environment 85

6.3.3 Application Detection on EC2 Cloud 91

6.3.4 Ping detection on EC2 91

6.4 Conclusion 93

7 Machine Learning based Website Detection 95

7.1 Motivation 95

7.2 Browser Profiling Scenarios 97

7.3 Website Profiling Results 100

7.4 Discussion 107

7.5 Outcome 109

8 Machine Learning based Side-Channel Attacks on Mobile Platforms 110

8.1 Motivation 110

8.2 Inference Attack 113

8.2.1 Attack Outline 114

8.2.2 Finding Eviction Sets . . . 114
8.2.3 Post-processing and Feature Vectors . . . 122
8.3 Experiment Setup and Results . . . 124
8.3.1 Target Device . . . 125
8.3.2 ML/DL Configuration . . . 125
8.3.3 Evaluation Results . . . 128
8.4 Discussion . . . 134
8.5 Outcome . . . 137

9 FortuneTeller: Machine Learning based Defense Mechanism 138
9.1 Motivation . . . 138
9.1.1 Methodology . . . 141
9.1.2 Implementation . . . 143
9.2 Evaluation . . . 147
9.2.1 Experiment Setup . . . 147
9.2.2 RNN Model Training . . . 148
9.2.3 Server Experiments . . . 150
9.2.4 Laptop Environment . . . 153
9.2.5 Optimizing the Sliding Window Size . . . 154
9.2.6 Prediction Time in Testing Phase . . . 155
9.2.7 Performance Overhead . . . 156
9.3 Comparison of *FortuneTeller* with Prior Detection Methods . . . 157
9.4 Discussion . . . 161
9.5 Outcome . . . 163

10 Conclusion 164

A **184**

A.1 List of the Websites 184

A.2 Additional Tables and Figures for Mobile Phones Attack 185

A.2.1 Profiled Applications, Websites, Videos 185

A.2.2 CNN Parameter Selection 187

A.3 Appendix for FortuneTeller 187

A.3.1 Tables for Performance Counters and Benchmarks 187

List of Figures

1.1 An overview of publications resulting from this dissertation work. The papers broadly contribute to cloud security, and use Machine Learning to automate microarchitectural attacks and attack detection techniques. 11

2.1 RNN, LSTM, GRU cells 24

4.1 Red and blue lines represent idle and RSA decryption/AES encryption access times respectively 38

4.2 The difference of clock cycles between base and RSA decryption profiling for each set-slice pairs over 10 experiments 39

5.1 Data access time in hardware cycles when the data is located in the cache and in the memory 45

5.2 Leakage Distributions f_0 and f_1 if Hypotheses H_0 and H_1 are correct. The measurements were taken in an Intel i5 2430M CPU in SSA scenario. 46

5.3 Comparison of the scores of key guesses in the natively executed FSA scenario for three different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c), applied to 10000 traces. The correct key is 180 and clearly distinguishable in all three cases. 54
5.4 Comparison of results in native execution for FSA scenario for different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c). 55
5.5 Comparison of results in native execution for the SSA for different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c). 56
5.6 Results in cross-VM execution for different attack scenarios using the miss counter distinguisher FSA (a) SSA (b) ASA (c) and the means distinguisher FSA (d) SSA (e) and ASA (f). 57
5.7 Different sets of data where we find a) trace that does not contain information b) trace that contains information about the key 65
5.8 10 traces from the same set where a) they are divided into blocks for a correlation alignment process b) they have been aligned and the peaks can be extracted 65
5.9 Eliminating false detections using a threshold (red dashed line) on the combined detection graph. 66
5.10 Comparison of the final obtained peaks with the correct peaks with adjusted timeslot resolution 67
5.11 Combination of two sets 68

6.1 The flow chart of the approach for both L1 and LLC profiling 74

6.2 Visualization of 10 core LLC. Gray set-slice pairs are noisy, white set-slice pairs are unused sets and black set-slice pairs are actively used by target application. 76
6.3 Eliminated noisy sets in LLC 80
6.4 One of the active sets for an application 80
6.5 Hit(0) and miss(1) graph of an active set 81
6.6 Frequency components of an active set 81
6.7 Combination of frequency components of all active sets 82
6.8 The scenario for ping detection on Amazon EC2 83
6.9 Success rate graph for varying number of sets to train the data 87
6.10 Success rate for different tests in L1-data (blue) and L1-instruction (yellow). The last bar represents the average of success rates for 40 tests 88
6.11 LLC success rate with varying number of frequency components . . . 90
6.12 LLC success rates for different tests in native scenario. The blue bar represents the success rate for different tests. The last bar shows the average success rate for all tests 90
6.13 LLC success rates for different tests in cloud scenario. The blue bar represents the success rate for different tests. The last bar shows the average success rate for all 25 tests 92
6.14 Cache miss pattern of received ping requests in LLC 93
6.15 Frequency components of ping requests in LLC 93

7.1 Success rates per website for (a) Google Chrome on ARM with the dashed line showing an average classification rate of 84%, and (b) Tor Browser on Intel with the dashed line showing an average classification rate of 68%. 101

7.2 Success rate vs. number of training measurements for Google Chrome (Incognito), and (a) 30 different websites (b) 10 same domain web pages. 102
7.3 Number of guesses vs. classification rate for (a) Google Chrome (Incognito), and (b) Tor Browser. Solid lines represent results for Alexa Top 30, while the dashed lines illustrate the same domain results. 103
7.4 Success rate vs. number of training measurements for the Tor Browser and (a) 30 different websites, or (b) 10 same domain web pages. . . . 104
7.5 (a) Success rate vs. number of training measurements for Tor Browser and all websites. (b) Number of guesses vs. classification rate for whistleblowing (dashed) and all websites (solid). 105

8.1 Mapping of virtual memory to cache sets. 113
8.2 Virtual/physical address and its interpretation. 114
8.3 Plots of (a) $t_{\mathcal{T}}$ and (b) t_p, as used in Algorithm 3. 118
8.4 Classification results for application inference over an increasing number of LLC profiles for (a) *ordered*, (b) *unordered*, and (c) *FFT* feature vectors. 127
8.5 Average receiver operating characteristic (ROC) curves for SVM, SAE, CNN during application inference. 129
8.6 Average receiver operating characteristic (ROC) curves for SVM, SAE, CNN during website inference. 129
8.7 Probability estimates from the CNN softmax layer while classifying known and unknown apps. 131
8.8 Website classification with our CNN for *ordered* (solid), *unordered* (dotted), and *FFT* (dashed) feature vectors. 131

8.9 Video classification with our CNN for *ordered* (solid), *unordered* (dotted), and *FFT* (dashed) feature vectors. 132
8.10 Average receiver operating characteristic (ROC) curves for SVM, SAE, CNN during video inference. 134
9.1 *FortuneTeller* implementation 142
9.2 Validation error with increasing number of measurements for Gnupg benchmark 149
9.3 Prediction error in Gnupg for LSTM algorithm 149
9.4 ROC curve for LSTM and GRU models in server and laptop environments 152
9.5 The validation error for different sizes of the sliding window 155

List of Tables

5.1 Distribution of cache accesses vs. memory accesses for the two hypotheses over the three attack scenarios. SSA provides the best distinguishability. 55
5.2 Successfully recovered peaks on average in an exponentiation 68

6.1 Symbol Descriptions 77

9.1 The False Alarm Rate in percentage per second for applications . . . 154
9.2 Comparison of previous methods 160

A.1 List of websites profiled PerfWeb. 184
A.2 List of profiled applications. 186
A.3 List of profiled websites. 187
A.4 List of profiled videos. 187
A.5 CNN parameter exploration. Final selection highlighted in bold. . . . 188
A.6 Counter Selection for core counters 189
A.7 Benchmark tests used in the experiments 190

Chapter 1

Introduction

Computers and smartphones play an important role in our lives as they enable many individuals to connect with the rest of the world. While people use these devices to access email accounts, social media, and workplaces, they also share their confidential information such as credit card information, passwords and even social security numbers with third-party applications and websites. Now that internet access has become widely available in all devices, usage of third-party applications has increased drastically in the last decade. Hence, many application providers started to build their infrastructure in cloud servers to handle intense application usage as well as to provide better security and safety for their customers.

For companies, security protocols have long been a part of standard business practices. The first action taken by companies is to teach employees the basic security protocols such as secure ways of using employees' own devices at work. Besides employee education, software and network security are also primary concerns to protect company's assets against cyber-attackers. While network security aims to protect incoming and outgoing network traffic, software security's purpose is to prevent confidential information leakage by providing software-based security

services. Both network and software infrastructure rely on the underlying hardware in which any design flaw could also be exploited through network and software.

Unfortunately, hardware security is of secondary importance since companies have considered physical access to devices as the main threat against hardware. Particularly with the increasing cloud resources usage, companies put the responsibility of hardware security on cloud computing providers. In order to provide a secure cloud environment, various virtualization methods such as hypervisors are implemented to isolate the cloud clients in the shared hardware architecture. Moreover, to enable high performance on cloud servers many features are integrated into hypervisors, which might potentially introduce new leakages that can be exploited by malicious users in public clouds.

Even though cloud service providers have the responsibility of hardware security, chip vendors such as Intel and AMD, which produce high performance architectures for computers and servers, also have a huge responsibility. At the same time, personal computers, smartphones and IoT devices also rely on their architecture. Therefore, any leakage introduced in the design process of an architecture could affect billions of devices worldwide. For this reason, chip vendors pay significant attention to designing secure architectures since it is not possible to change the design after production. However, increasing complexity of microarchitectures makes identifying potential leakages challenging. Thus, leakage sources must be identified to improve the level of security before the cyber-attackers exploit them. Moreover, every company and client introduce various modifications to hardware design by configuring the hardware features as well as installing third-party libraries, and operating systems. All these modifications might weaken the device security, which makes it essential to analyze the devices again by considering new threat models. Since it is almost impossible to identify the leakages manually for a large number

of designs, automated analysis techniques such as Machine Learning (ML) algorithms can be employed. In the following sections, a brief overview of cryptography, hardware security and Machine Learning will be given.

1.1 Cryptography

The implementations of cryptography evolved to symmetric and asymmetric cryptosystems where the same secret key is used in both encryption and decryption operations in symmetric systems such as AES (Advanced Encryption Standard) and DES (Data Encryption Standard). On the other hand, asymmetric systems involve distinct keys for encryption and decryption such as RSA (Rivest-Shamir-Adleman), ECC (Elliptic Curve Cryptography). Both systems provide sufficient defense mechanisms against mathematical attacks so that they are used in daily applications to provide software security.

Even though the cryptographic algorithms establish efficient defenses in theory, the software implementations would be vulnerable to hardware-based attacks. In general, the developers try to produce bug-free implementations to verify the correctness of the cryptographic algorithms. However, it was not expected to introduce countermeasures in cryptographic implementations against hardware related attacks until Kocher et. al. [99] showed that the timing information leaks the secret keys. This novel attack started a new research area in cryptography which shows that the cryptographic algorithm implementations may leak confidential information due to the underlying hardware flaws.

1.2 Cloud Security

Cloud providers offer an array of computing sources for both small and large companies which race against each other to keep up with the increasing demand from customers. The transition from local servers to cloud servers began only a decade ago and 90% of the companies use hosted cloud service in 2019 [8]. The main factors of the fast transition are flexibility, reliability and security provided by the cloud companies. All the requested services are handled by cloud providers like maintaining and updating systems as well as giving opportunities to fulfill the core business strategies. Therefore, the cloud services are under high demand, thus, it is expected that cloud data centers will process 94% of workloads in 2021 [4]. Depending on the type of request from the companies, cloud providers offer 4 different cloud services. Software as a Service (SaaS) is used by the clients who need to access their data in the applications over the internet. Platform as a Service (PaaS) is used to develop and test new applications before releasing to the customers. Machine Learning as a Service (MLaaS) offers many Machine Learning applications to the customers who are willing to develop new ML algorithms and use cases. Finally, Infrastructure as a Service (IaaS) cloud offers a vast array of computing sources in a virtualized environment which includes servers, networking and data storage. It is predicted that IaaS spending will be the fastest growing category among 4 services in the following 5 years [12].

The privacy and security issues in the cloud environment worry more than 66% of the IT professionals [47]. These issues mostly arise from both outdated software tools such as usage of vulnerable cryptographic libraries and design flaws in the underlying hardware. Therefore, the cloud providers are supposed to follow the patches released by both software and hardware vendors to establish more secure cloud environment

for their clients. However, it is not sufficient all the time since cyber-attackers always discover new ways to leak confidential information from public clouds. Throughout my Ph.D. studies, I focused on the side-channel techniques to exploit the leakages in the shared hardware resources on public cloud servers.

1.3 Microarchitectural Attacks

Side-channel attacks rely on any additional information gathered from the design of a system. Power consumption, timing, electromagnetic or acoustic information of a specific operation would give side-channel information which can be collected by an antenna, magnetic probe or an oscilloscope. Then, the collected measurements are analyzed to extract the secret information. While some of these techniques require to have physical access to the targeted device, micro-architectural based side-channel attacks can be built on a remote access to the device.

Microarchitectural attacks pose a significant threat against shared hardware resources. Starting with Kocher [99], many cryptographic implementations have become targets for the timing-based attacks. This novel attack is capable of observing the individual operations for each secret key bit. The high resolution-based profiling enables attackers to extract the secret key partially by observing runtime differences for different operations and then, partial information obtained from each bit is combined to construct the entire secret key. Hence, Kocher [99] demonstrated that the mathematical strength of brute force on cryptographic implementations can be circumvented by implementing side-channel attacks. To understand the root cause of runtime differences, researchers started analyzing hardware optimization techniques. The first issue was determined as the weaknesses in the cache structure, which yielded to various cache attacks [128, 132, 172] in the past. The demon-

strated cache attacks are based on both access time difference between cache and main memory, and non-constant time implementations of cryptographic algorithms. In addition, cache attacks were demonstrated on various targets other than cyrptographic key extraction [26, 85, 171] such as keystroke recovery [107] and browser profiling [126]. All these attacks indicate the potential destruction of people's security and privacy through cache usage.

In 2018, another class of microarchitectural attacks namely, transient execution attacks were introduced. Spectre [97] and Meltdown [109] rely on speculative and out-of-order executions, respectively, integrated into the hardware design by the chip designers. The attacks also leverage cache attack techniques to extract secret information, which demonstrates the importance of cache structure in secret information recovery. The exploited optimization techniques were present in both Intel and AMD devices for a long time however, they seemed secure against microarchitectural attacks until Spectre and Meltdown were introduced. Since many attacks rely on various hardware optimization techniques, some leakages stay hidden for years until an expert analyzes the specific features deeply. The analysis takes countless hours of reverse engineering and identifying the weakness in the software/hardware implementations of targeted applications. Therefore, many leakages exist in the hardware implementations and they are still not explored due to the required intensive labor. In order to enable the automated leakage detection and side-channel information analysis we leverage Machine Learning techniques in our works.

1.4 Machine Learning and Deep Learning

In side-channel attacks, the analysis of weaknesses in both software and hardware implementations is done by the humans. Especially, with the increasing number

and diversity of platforms, it becomes more challenging to identify potential leakage sources. In order to ease the heavy lifting on side-channel analysis experts, we propose Machine Learning and Deep Learning based solutions to automate the feature extraction and classification of the side-channel data.

Machine Learning (ML) algorithms are implemented to increase the automation in various tasks. The main purpose is to automatically learn certain tasks by observing data without any human intervention or assistance. The first appearance of ML was in 1960s where pattern classification was the primary focus of ML research [125]. Next, Artificial Intelligence (AI) became popular by focusing on having machines learn from data. Moreover, the theoretical emphasis on the knowledge-based approach increases the attention on AI systems. Especially, after the neural networks were introduced, researchers realized the potential of AI in many tasks. Finally, in 1990s, machine learning is recognized as a separate field which aims to solve problems in a practical nature.

Even though ML algorithms were designed to increase the efficiency in automation, the capabilities of ML were limited due to the lack of huge datasets and computational power. Therefore, until the GPU systems were improved in terms of computing capabilities, ML was mostly applied to small tasks. Next, the researchers started improving the algorithms and deep neural networks [55] became the most popular learning technique as the success rate of Deep Learning (DL) has achieved human level. Then, the integration of DL algorithms into our lives started. In many applications, DL algorithms surpassed humans such as playing games [20, 53], detecting the diseases [131], self-driving cars [14] and so on. Motivated by the Deep Learning capabilities on many challenging tasks, we aim to integrate the capabilities of DL algorithms into side-channel analysis. We mainly aim to solve most common problems such as mis-alignment, system noise, feature extraction in the side-channel

measurements as well as increasing the success rate of classification tasks without any expert involvement.

1.5 Problem Statement

Side-channel attacks pose a serious threat against security and privacy of cloud, personal computer, and mobile phone users. Since each attack technique targets complex hardware components, cryptographic libraries or third-party applications, tremendous effort is required to verify that such systems are free from any vulnerabilities. Due to the increasing diversity of devices as well as hardware/software complexity, large-scale vulnerability analysis becomes more challenging than before.

1.6 Thesis

We hypothesize that Deep Learning (DL) techniques can be used to mitigate the scalability problem by automating large-scale vulnerability discovery across platforms. This thesis is motivated by the fact that, DL techniques have proven to be effective in large-scale and complex automation such as in computer vision, text/speech processing, self-driving cars, medical diagnosis, industrial design and in game development. To test this theory, an analysis of potential security and privacy leakages on cloud and mobile platforms has been automated by processing a large amount of side-channel data with advanced DL systems. Throughout this thesis, we demonstrate that DL techniques have the capacity to automate the vulnerability discovery by figuring out features without expert intervention. We show that DL-based vulnerability analysis techniques can discover underlying leakages in third-party applications as well as hardware designs. This thesis only provides an initial evidence by showing that applications of DL algorithms on massive-scale vul-

nerability discovery are viable. Since the thesis focuses on leakages in the underlying hardware, in the future work, NLP-based and generative DL techniques should also be investigated to improve the large-scale vulnerability analysis on other domains such as software-based vulnerabilities.

1.7 Contributions

In this dissertation, we investigate the potential for Deep Learning-assisted microarchitectural attacks to enable large-scale exploitation by targeting the security and privacy of users as well as prevention of such exploits across cloud computing resources, mobile phones and personal devices. The overview of the contributions is given in Figure 1.1.

Contributions can be summarized as follows;

- We examine the possibility of achieving co-location in popular public clouds such as Google Compute Engine (GCE) and Amazon EC2 cloud. We show that by profiling Last Level Cache (LLC) sets during cryptographic operations, it is viable to detect the co-located VMs in the cloud.

- We implement a faster version of Flush+Reload attack in virtualized environments by exploiting the memory deduplication feature. A secret AES encryption key belonging to a third-party user is recovered by collecting a few encryption measurements. This study shows that an AES encryption process can be detected automatically, which does not require a synchronization between attacker and victim VMs in the cloud.

- We show that Prime+Probe attack is applicable in the public clouds such as Amazon EC2. We recover full 2048-bit RSA key from another VM by

monitoring active cache sets in LLC. In order to reduce the public cloud noise in the measurements, we develop a correlation-based analysis technique which enables the attackers to recover entire RSA decryption key.

- We demonstrate that LLC attacks can be an offensive tool against customers' privacy in the public clouds. We develop a generic LLC profiling tool which is based on ML algorithms to identify the running applications in the co-located VMs as well as detecting the co-located VMs with ping requests.

- We introduce a new technique called PerfWeb which is based on performance counter profiling to violate the web privacy in the personal computers. Our tool is improved with advanced Deep Learning algorithms to increase the performance of website classification. Furthermore, we show that privacy protecting browsers such as Tor browser are not efficient against PerfWeb.

- We analyze the applicability of ML-based cache attacks on mobile platforms. We discover better methods to implement Prime+Probe attack on commonly used mobile phones. Our tool achieves high success rate in identifying the running applications, browsers and even videos in a third-party mobile phones.

- Finally, we propose a DL-based unsupervised detection mechanism called FortuneTeller to detect micro-architectural attacks in both personal computer and server environments. We leverage superior capabilities of advanced RNN algorithms such as Long-short term memory (LSTM) and Gated Recurrent Units (GRUs) to predict the subsequent performance counter values during benign application processes. We show that a wide range of microarchitectural attacks can be detected with a high accuracy.

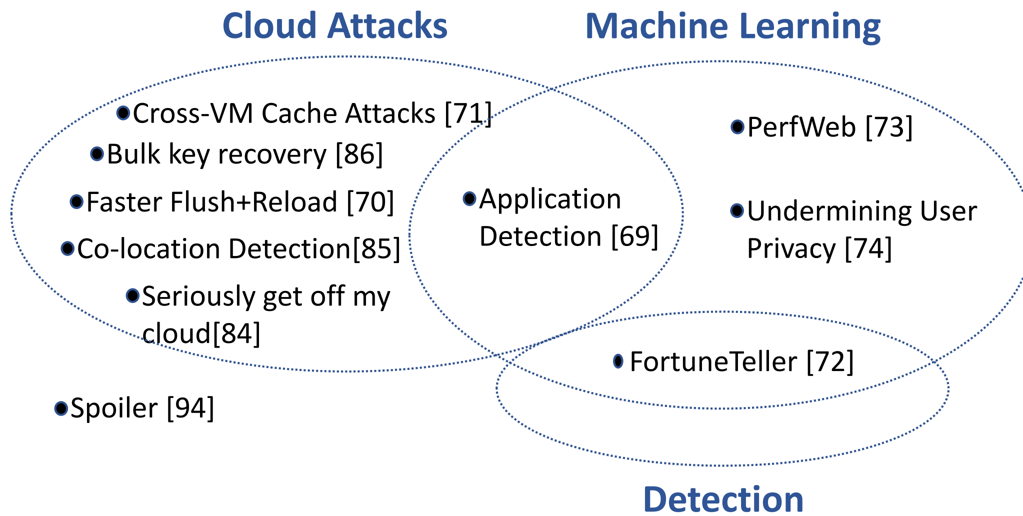


Figure 1.1: An overview of publications resulting from this dissertation work. The papers broadly contribute to cloud security, and use Machine Learning to automate microarchitectural attacks and attack detection techniques.

1.7.1 The publications resulted in this dissertation

The research described in this dissertation is the result and combination of the following peer-reviewed publications on various conferences and journals [67, 81, 83, 68, 82, 66, 70, 71, 69, 176]. The publications [67, 81, 83, 68] are the result of the collaborative work with Mehmet Sinan Inci and Gorka Irazoqui. The outcomes presented in Chapter 7 and Chapter 8 are the results of collaborative work with Andreas Zankl published in [70, 71, 176]. The results presented in Chapter 9 are the collaborative work with Daniel Moghimi [69].

Chapter 2

Background

In microarchitectural attacks, attackers need to co-locate with a victim through a covert channel. While in a cloud platform, it might be a co-location with a victim in the same hardware but in different virtual machines, in mobile phones and personal devices, third-party applications would execute malicious code snippets. It is commonly assumed that the attacker is not capable of changing any configuration in the device since the malicious application has no privilege. The attacker targets cryptographic libraries to recover secret keys as well as monitors user activity in the same machine. Hence, the secret dependent operations in the software are targeted by the attackers. Especially, non-constant time operations on secret information is subject to reveal the secret to the attackers through microarchitectural components. As each application has different footprint in the components, attackers can also identify the actions taken by the user in the device. Once the attacker collects the side-channel data through covert channel, the data needs to be analyzed to construct the entire secret or the action.

This dissertation focuses on application of microarchitectural attacks on a variety of targets as well as mounting ML algorithms to analyze the large amount

of dataset. Before the explanations of proposed attacks, a brief background on microarchitectural attacks and implemented ML algorithms are given.

2.1 Computer Architecture

This section explains the details of memory structure, CPU cache and cache attack techniques which are essential for implementing a cache-based profiling technique.

2.1.1 Memory Hierarchy

The memory design of a computer system consists of primary and secondary memory. The primary memory includes registers, cache and main memory which can be accessed by the processor directly. The secondary memory is accessible through an input/output module which includes external memory resources such as magnetic or optical disk. While a program is executing, the memory contents are loaded from the hard-disk to the Dynamic Random Access Memory (DRAM). The loaded content from DRAM is also brought to the cache and registers since the access time gets lower when the data resides in upper levels in the memory hierarchy for the next accesses to the same content. As a result, registers and CPU cache offer fast access to the content when it is requested later by the same program.

Memory Deduplication

Memory deduplication is an OS memory optimization technique that allows the OS to keep only a single copy of a data in the memory when multiple processes are using the same data. While this feature is useful in native execution, it is even more useful in virtualized setting where many VMs use the same OS and/or the same software.

Hence, to reap the benefits of the deduplication, VMMs have also implemented

memory deduplication techniques to allow more VMs to run on the same physical machine. For this, the VMM recognizes identical and redundant memory copies by first checking their hash values and then performs a bit-by-bit comparison. If the memory content is determined to be shared by more than one process/VM the memory manager removes multiple copies from the memory. Note that even though this deduplication process is only performed on shareable memory pages like shared libraries, shared libraries are used in many software packages. Memory deduplication methods are especially effective when hosting multiple processes, as is the case in virtualized systems. Consequently, VMMs like VMware [161, 162] and KVM [13, 95] implement variations of memory deduplication, i.e. Transparent Page Sharing(TPS) and Kernel Samepage Merging(KSM), respectively. While the memory saving optimization techniques improve the performance they also create a covert channel that a malicious VM can exploit. In fact, memory disclosure attacks [149] and side-channel attacks [172, 25, 89] have been proposed taking advantage of memory deduplication techniques in the cloud.

2.1.2 CPU Cache

CPU cache is a small memory structure between main memory and CPU cores, which is used to access the content faster than main memory by CPU. Cache is divided into levels (L1,L2,L3 and L4) where upper levels are smaller than the lower levels but the access time is faster. In L1 cache, instruction and data caches are separated however, other levels store both instruction and data types. The data transfer between memory and cache is performed in blocks of a fixed size, called *cache lines*. If a data resides in a portion of a cache line, the entire cache line is loaded from memory to the cache. Each cache consists of sets and ways where cache lines are stored in the ways. The mapping function from virtual memory to cache

sets varies among processors. While L1 cache is virtually indexed, physically tagged, other levels are physically indexed and tagged. Therefore, it is not possible to know which memory address is mapped to a specific cache set by only using the virtual address number.

When a cache set is fully occupied i.e. all the ways are filled up with cache lines, a pre-determined replacement policy is applied to decide on which cache line will be replaced with the new cache line. The most common policy is least-recently used (LRU) which evicts the least recently accessed cache line from the cache set. While LRU is the most common policy among Intel processors, ARM processors mostly implement a random replacement policy which evicts a random cache line from the cache set.

2.1.3 Cache Attacks

In this section, two common cache attack techniques namely, Prime+Probe and Flush+Reload, are described. Both attack techniques have been used to demonstrate the threats against cloud security in our publications.

2.1.3.1 Flush+Reload Cache Attack

Flush+Reload attack is a trace driven cache side-channel attack that was first used by Gullasch et al. [65]. Then, a high resolution version of Flush+Reload attack [172] was proposed. The attack is based on shared memory leakage due to the memory deduplication process.

Flush+Reload Technique

Flush stage:

In this stage, the attacker flushes one or more of the desired memory locations from the cache by using the `clflush` command which evicts the desired memory locations from the entire cache hierarchy. The eviction is implemented on the entire cache structure even though the memory address is in another core's cache. Indeed, this is the main reason why the attack is applicable across cores even in the cloud environment.

Wait stage:

In this stage, the attacker waits for sufficient time for the victim to use (or not use) the memory locations that he has flushed in the previous stage. The flushed memory addresses mostly contain secrets or secret related data/instructions which are targeted by the attacker.

Reload stage:

In the final stage, the attacker reloads the previously flushed memory locations while measuring the reload time for each of them. If the victim accessed one of the flushed memory lines, due to the inclusiveness of the shared last level cache, the memory address is loaded in both upper level caches and shared last level cache. Thus, the attacker will measure a lower reload time compared to data accesses from the main memory since the cache line will be retrieved from the cache. However, if the victim did not access to the data flushed in the first stage, the data will still reside in the memory, causing a higher reload time in the reload stage.

2.1.3.2 Prime+Probe Cache Attack

In the modern computer architecture, it is not possible for users to see the physical address of a line because of the security issues. Therefore, the virtual address is translated from the physical address and it is visible to users. In virtual address the first 12 bits are exactly same with the first 12 bits of physical address. However, this is not enough to find the corresponding cache set for the line in LLC. Thus, it is not possible to create an eviction set with regular 4KB pages in LLC by using virtual address.

Prime+Probe technique is the most widely applicable profiling technique on the cloud since all major Cloud Service Providers (CSPs) have disabled deduplication, making Flush+Reload attacks infeasible. To create an eviction set in LLC, the spy needs to know more than 12 bits of the physical address. When Huge pages (2MB) are allocated by the attacker, the first 21 bits (2MB = 21 bits) are converted from virtual to physical address directly. As the corresponding LLC set number of a memory address is based on the first 21 bits, the attacker can determine the memory addresses for the targeted LLC set. Therefore, the attacker creates the eviction set by filling the LLC set with the corresponding memory addresses. After finding the eviction set for the targeted set, the eviction process can be implemented. The Prime+Probe profiling is divided into three main stages as follows:

Prime stage: This stage is used to create an eviction set. To create an eviction set the spy generates distinct lines which reside on the monitored set. The number of lines in the eviction set is equal to number of ways in the monitored set. After all lines accessed by the spy the eviction set is ready.

Waiting stage: In this stage, the spy waits for the target to evict some lines from the primed set. The waiting time is crucial to determine the resolution of the profiling. While the time is increasing the frequency and resolution are getting

lower.

Probe stage: In the probe stage, the spy accesses the addresses used in the prime stage. If the monitored set was not accessed by another process, no data has been evicted; all accesses result in a cache hit, giving a low access time. If another process has accessed the monitored set, its data must have evicted at least one of the lines of the spy's data. Hence, the probe access will include accesses to memory, resulting in measurably higher access times.

In Intel architectures there are two types of cache slice selection algorithm namely, non-linear and linear slice selection algorithm. In linear slice selection algorithm, same lines can be used to create an eviction set by changing the set number of the line. However, in non-linear slice selection algorithm for each set the eviction set should be created by implementing the algorithm. This makes the process harder because to find the eviction set for all sets in LLC by hand takes huge amount of time. Therefore, the algorithm in [68] is implemented in 10 core machine to create LLC eviction sets faster.

2.2 Hardware Performance Events (HPEs)

Counting HPEs is supported by a large spectrum of modern processors. Typically, each core implements a performance monitoring unit (PMU) that is responsible for counting events. The unit implements a set of hardware counters that can each be configured to count events of a certain type. Often, the number of available events is considerably larger than the number of available counters. Consequently, only a limited number of events can be counted in parallel. In order to measure more events, software layers that use the PMU implement time multiplexing. This works by frequently re-configuring counters and reading event counts only for a limited

period of time. As a result, this yields pseudo-parallel event counts.

Access to PMUs is typically restricted to privileged, i.e., kernel or system level code, but interfaces exist through which user space applications can gather event counts. Interfaces can be tailored to operating systems, e.g., on Windows [120], or defined more generically, e.g., `PAPI` [105] and `perf` [106]. In this work, we focus on the `perf` interface that is mainly found on Linux systems. Note that this work demonstrates the general feasibility of website fingerprinting with HPEs. Therefore, similar results are also expected on systems with other HPE interfaces.

2.2.1 Profiling with Perf

The `perf` event monitoring subsystem was added to the Linux kernel in version 2.6.31 and subsequently made available to the user space via the `perf_event_open` system call. The `perf_event_attr` is the main configuration object. It determines the type of event that should be counted and defines a wide range of acquisition properties. We focus only on a very limited number of settings and use zero values for all others. This renders our measurements to be reproducible on a larger number of systems. The `type` field in `perf_event_attr` specifies the generic event type. As we focus on hardware based events, we only use `PERF_TYPE_HARDWARE` or `PERF_TYPE_HW_CACHE`. The `config` field determines the actual event type. The event selection used in this work is given in Section 7.2. In addition, we set the `exclude_kernel` option, which avoids counting kernel activity. This improves the applicability of our measurement code, because kernel profiling is prohibited on some systems. Finally, the `size` field is set to the size of the event attribute struct. The `pid` and `cpu` parameters are used to set the scope of the event profiling. In this work, we focus on two profiling scenarios: process-specific and core-wide. To limit event counting to a single process, `pid` is set to the process identifier and `cpu` is set to `-1`.

Subsequently, events are counted only for the given process, but on any processor core. To enable core-wide counting, `cpu` is set to the core number that should be observed and `pid` is set to `-1`. Events are then counted only on one processor core, but for all processes running on it. The `group_fd` parameter is used to signal that a selection of events belongs to a group. The `perf` system then counts all members of a group as a unit, i.e., their values can be meaningfully compared. This is not a requirement for our approach, as all measured performance events are viewed as separate inputs to the machine learning algorithms. Thus, we omit `group_fd` and set it to `-1`. The `flags` parameter is used to configure advanced settings including the behavior when spawning new processes and monitoring Linux control groups (cgroups). As none of the settings are relevant to our measurement scenarios, we set `flags` to zero.

Once `perf_event_open` succeeds, the returned file handle can be used to read and reset event counts as well as to enable and disable counting. In our measurements, we read event counts using the standard `read` system call. We found this to yield a sufficiently high sampling frequency and subsequently high success rates during website fingerprinting. On our test systems, the duration of the `read` system call ranges between 1.5 μs and 3.0 μs when reading one counter value.

2.2.1.1 Access Control

On Linux, access to HPEs can be configured for user space applications. The access level is specified as an integer value that is stored in `/proc/sys/kernel/perf_event_paranoid` in the `procfs` filesystem. A negative value grants user space applications full access to performance profiling. If the `paranoid` level is set to `0`, comprehensive profiling of the kernel activity is prohibited. A value of `1` prevents user space applications from core-wide event counting (`pid = -1`, `cpu > 0`). A `paranoid` level of `2` prohibits

process-specific event counts while the application gives control to kernel space, e.g., during a system call. Values above `2` deny event counting even in user space and essentially deactivate `perf` for user space applications. Note that the `paranoid` setting is typically overridden by applications started with the `CAP_SYS_ADMIN` capability, e.g., programs started by the root user.

2.3 Machine Learning Techniques

Machine Learning (ML) provides powerful tools to automate the process of understanding and extracting relevant information from noisy observations. ML is divided into three categories namely, supervised, unsupervised and reinforcement learning. The supervised techniques are mostly implemented in the scenarios where the labels of each input are given to the ML model. On the other hand, unsupervised techniques require no labeling of the input where the aim of the ML model is to learn the pattern of the input itself. The reinforcement learning is mostly implemented in a dynamic environment where the model receives feedback from the environment depending on the actions taken by the model. In this dissertation, both supervised and unsupervised techniques have been employed to improve the side-channel data analysis in microarchitectural attacks and defenses.

The collected data is separated into training, validation and test datasets. While training and validation datasets are used to train the ML model, the accuracy of the model is measured by the success rate on the test data. In supervised learning, the success rate of a ML technique denotes the percentage of unknown samples that are classified correctly. To reliably determine the success rate, classification is performed multiple times with different training and test sets that are derived through statistical sampling. This so-called cross-validation is performed, if the

number of overall samples is low.

Anomaly detection is one of the main application areas of unsupervised learning where the anomaly in the data samples is detected by the ML model. The model is trained with benign data samples and then, model's accuracy is calculated by feeding benign and anomalous activity data. Once the model can separate the benign and anomalous data with a higher success rate, the training of the model stops. Further details of the Machine Learning techniques used throughout this dissertation are given in the following paragraphs.

k-th Nearest Neighbor (kNN). The main goal of kNN is to find a training sample that is closest to a test sample according to the Euclidean distance. The smallest distance is taken as the first nearest neighbor and the test sample is marked with the corresponding label. In our experiments, we use the *fitcknn* command to implement kNN and to train our models. By default, the prior probabilities are the respective relative frequencies of the classes in the data, which are initially set to be equal to each other.

Decision Tree (DT). Decision Trees are used to classify samples by creating branches for given data features that yield the best split among all classes. The values for each branch are chosen such that they minimize the entropy. In our experiments, we use the *fitctree* command to train the model. The default value for maximum split is $N-1$, where N denotes the number of classes. For the training phase, the minimum leaf size is 1 and the minimum parent size is 10.

Support Vector Machine (SVM). In SVM based learning, input data is converted to a multi-dimensional representation by using mapping functions. Hyperplanes are then created to classify the data. The general strategy is to find the optimal decision boundaries between classes by increasing the distance between

them. In our experiments, we use `libsvm` [34] to implement multi-class Support Vector Machines. The model is created and trained based on a linear SVM. We set the type of the SVM to C-SVC, where the parameter C is used to regularize the mapping function.

Convolutional Neural Network (CNN). In contrast to the other ML techniques, Convolutional Neural Networks automatically determine important features of the input data. This is achieved by creating nodes between higher and lower dimensional input data mappings. The meaningful features are then extracted by finding the optimal functions for each node. In our experiments, we choose two autoencoders to classify our measurements into N classes. In each autoencoder, different levels of abstraction are learned from the feature vectors and mapped to a lower dimensional space. While the number of layers in the first autoencoder is $100 \cdot N$, the second autoencoder has $10 \cdot N$ layers. The maximum number of iterations is set to 400 and L2 weight regularization is set to 0.001 for both autoencoders. The last layer is the softmax layer. The training data is trained in a supervised fashion using labels. After the neural network is established and first classification results are obtained, the accuracy of the multilayer network model is improved using backpropagation and repeated training using labeled data. While CNNs have many advantages, the most important disadvantage is their memory demands. When we run out of GPU memory, we downsample the input data to reduce the length of the feature vectors.

Recurrent Neural Networks (RNNs). RNNs are a type of Artificial Neural Network algorithm, which is used to learn and predict the sequential data. RNNs are mostly applied to speech recognition and currently used by Apple's Siri [146] and Google's Voice Search [36]. The reason behind the integration of RNNs into real-world applications is that it is the first algorithm to remember the temporal

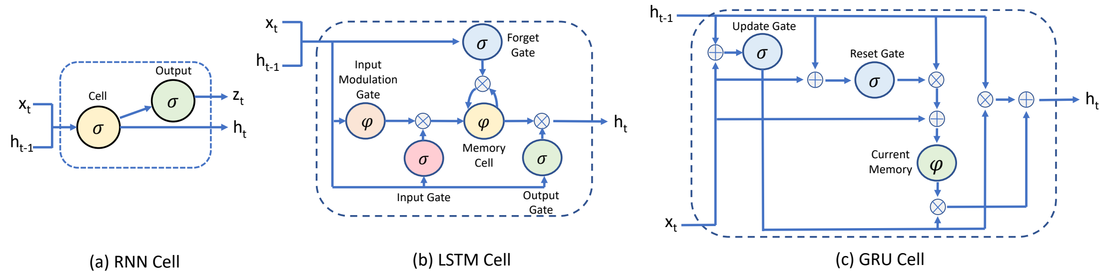


Figure 2.1: RNN, LSTM, GRU cells

relations in the input through its internal memory. Therefore, RNNs are mostly preferred for tasks where sequential data is involved.

In a typical RNN structure, the information cycles through a loop. When the algorithm needs to make a decision, it uses the current input x_t and hidden state h_{t-1} where the learned features from the previous data samples are kept as shown in Figure 2.1a. An RNN algorithm produces output based on the previous data samples and provides the output as a feedback into the network. However, traditional RNN algorithms are not good at learning the long-term sequences because the amount of extracted information converges to zero with the increasing time steps. In other words, the gradient has vanished and the model stops learning after long sequences. To overcome this problem, two algorithms were introduced, as described below:

Long-Short Term Memory. Long-Short Term Memory (LSTM) networks are modified RNNs, which essentially extend the internal memory to learn longer time sequences. LSTM network cells consist of a memory cell, input, forget and output gates as shown in Figure 2.1b. The memory cell keeps the learned information from the previous sequences. While the cell state is modified by the forget gate, the output of the forget gate multiplies the specific positions in the input matrix by 0 to forget and by 1 to keep the information. In the input gate, the useful information

sections are determined to be fed into the cell state. The activation function for input modulation gate is in general, *tanh*. Finally, the output gate passes the output to the next hidden state As a result, LSTM networks can select distinct features in the time sequence data more efficiently than RNNs, which enables learning the long-term temporal relations in the input.

Gated Recurrent Unit. Gated Recurrent Unit (GRU) is an improved version of RNNs. GRU uses two gates called, update gate and reset gate. Both current input and the previous hidden state are multiplied with their weights and added together in update gate. Then, a sigmoid activation function is applied to map the data to values between 0 and 1. The role of the update gate is to determine the amount of the past information to be passed along to the future. Next, the reset gate is used to decide how much of the past information to forget. The element-wise product between the reset gate and weighted previous hidden layer state determines the information to be removed from previous time steps. Finally, the current information is calculated The purpose of this part is to use the information obtained from the update gate and combine both reset and update gate information. Hence, while the relevant samples are learned by update gate, the redundant information such as noise is eliminated by the reset gate.

In this work, the RNN algorithms are used in an unsupervised fashion where there is no need for separate validation dataset in the training phase. The validation error is calculated for each prediction in the next timestamp and the total validation error is given after each epoch.

Chapter 3

Related Work

Our work consists of micro-architectural attacks, co-location detection, website fingerprinting and defense mechanisms against micro-architectural attacks. For each subject the corresponding related work is given in the following sections.

3.1 Co-location Detection

In the last few years several methods were proposed to detect co-location on commercial clouds [134, 180, 23, 183, 81]. These works use methods such as deducing co-location from instance and hypervisor IP address, hard disk drive performance degradation, network latency and L1 cache covert channel. However, in response to these works, most of the proposed techniques have been closed by public cloud administrators. Later Zhang et al. [180] were able to determine whether a particular user's VM had someone else co-residing in the same physical core. In particular, they utilized the well known Prime+Probe cache based side-channel technique to guess this information. However, the technique was applied in the upper level caches, thereby limiting its applicability to a physical core rather than the entire CPU or the machine. Furthermore, the technique was not tested in commercial clouds.

Shortly later, Bates et al. [23] demonstrated that a malicious VM can inject a watermark in the network flow of a potential victim. In fact, this watermark would then be able to broadcast co-residency information. Again, even though the technique proved to be extremely fast (less than 10 seconds), it was never tested in commercial clouds. Recently, Zhang et al. [183] demonstrated that Platform as a Service (PaaS) clouds are also vulnerable to co-residency attacks. They used the Flush+Reload cache side-channel technique together with a non-deterministic finite automaton method to infer co-location with a particular server. The technique proved to be effective in commercial PaaS clouds like DotCloud or OpenShift, but would never work in IaaS clouds where the memory de-duplication is not implemented, as in most of the commercial IaaS clouds.

Finally, Inci et al. [81] demonstrated that many of the previously utilized techniques in [134] are no longer exploitable. Nevertheless, they prove to detect co-location *across cores* in Amazon EC2 by monitoring the usage of the LLC with the Prime+Probe technique. To enable the co-location test, the authors make use of hugepages commonly available in commercial clouds. This feature provides a large memory space for the attacker to move and hit necessary addresses to prime cache sets. Also in 2015, Varadarajan et al. [158] investigated co-location detection in public clouds by triggering and detecting performance degradations of a web server using the memory bus locking mechanism. Simultaneously Xu et al. [170] used the same memory bus locking mechanism to explore co-location threat in Virtual Private Cloud (VPC) enabled cloud systems.

3.2 Microarchitectural Attacks

A typical microarchitecture has many components to increase the performance of a system. Each component would be subject to a micro-architectural attack. In the following paragraphs, relevant microarchitectural attacks are explained:

Cache Attacks While L1 and L2 cache are shared within the core, L3 cache is shared among all the cores in the CPU. When a cache attack targets L1 or L2 cache, Simultaneous Multithreading (SMT) feature is exploited where multiple independent threads can share the same features in a core. For instance, Percival [132] exploited L1 and L2 caches by applying Prime+Probe attack to recover RSA keys from OpenSSL 0.9.7c's implementation. In a similar way, Acıiçmez [16] targeted L1 instruction cache to distinguish square and multiply operations in an RSA decryption process. Later, Acıiçmez et al. [17] extended his previous work to DSA implementation of OpenSSL 0.9.8I. Tromer et al. [152] and Osvik et al. [128] also implemented Prime+Probe attack to L1 data cache to steal AES keys.

Even though L1 and L2 cache attacks are less noisy and faster due to the low number of cache sets and ways, they are not applicable for cross-core attacks. On the other hand, L3 cache-based attacks can be implemented to exploit confidential information from other processes running in another core. The Flush+Reload attack [171, 172] was used to steal cryptographic keys from RSA and ECDSA implementations. Irazoqui et al. [89] showed that AES keys can be recovered in a virtualized environment by exploiting page deduplication feature as well as running cryptographic libraries [90]. On the other hand, Prime+Probe attack on L3 cache requires the knowledge of physical bits of an address, which makes the attack more challenging. In 2015, Irazoqui et al. [85] applied a Prime+Probe attack on L3 cache by allocating 2MB huge-pages which give partial information on physical address.

The implementation of Flush+Reload, Evict+Reload and Prime+Probe attacks was also demonstrated on mobile phones [107] to recover AES keys and keystrokes.

Transient Execution Attacks The speculative and out-of-order executions are integrated into chip designs to optimize the performance. If some instructions are executed speculatively, the results are not committed to registers until the condition on the branch is resolved. When there is a mis-speculation a roll-back is implemented so that wrong results are discarded. However, after the roll-back the microarchitectural states are not cleaned which would cause secret leakage.

In the first Spectre study [97], two variants were introduced. While Spectre-V1 exploits the conditional branch prediction mechanism when a bound check is present, Spectre-V2 manipulates the indirect branch predictions to leak the secret. Next, researchers discovered new variants of Spectre-based attacks. For instance, a variant of Spectre focuses on poisoning Return-Stack-Buffer (RSB) entries with the desired malicious return addresses [100, 114]. Another variant of Spectre called "Speculative Store Bypass" [74] takes the advantage of memory disambiguator's prediction to create leakage. Similarly, Islam et al. [91] leverages the speculative execution of dependent load-store operations to leak a part of physical address from user-space applications, which can be used to boost Rowhammer and cache attacks. Then, researchers showed that there are also other covert channels than cache covert channel to measure the time difference: namely, using network latency [140], port contention [28], or control flow hijack attack based on return-oriented programming [116].

Meltdown [109] exploited out-of-order execution feature to reach secrets in kernel. Then, the work was extended to leak data from line-fill buffer [138, 156], load and store buffers [121]. The Meltdown-based attack techniques [153, 154] were also implemented in SGX environment.

3.3 Other Microarchitectural Attacks

Other than cache structure, several microarchitecture components were targeted. For instance, Translation Look-aside Buffer (TLB) was exploited to attack on libgcrypt EdDSA. TLBleed [57] implements a Prime+Probe-based attack on L1dtlb and STLB which are shared within a core. Another popular target is branch prediction units which are used to predict the outcome of a conditional branch or a jump instruction. Evtyushkin et al. [44] leverages directional branch predictor to simplify the direction recovery. Other than constructing a side-channel, Evtyushkin et al. [43] presents a Branch Target Buffer (BTB) based attack to bypass the ASLR protection. Furthermore, Aldaya et al. [18] exploits different ports in the execution engine in a SMT attacker model to leak the cryptographic keys.

Rowhammer is a type of microarchitectural attack which aims to flip the bits in DRAM to gain a privileged mode or achieving a fault injection. First, Seaborn et al. [141] showed that bit flips can be triggered by using *clflush* instruction to gain kernel level privilege. Then, the applicability of Rowhammer attacks was demonstrated in a cloud environment [169], Javascript [61] and mobile phones [155]. Recently, Frigo et al. [48] applied a Rowhammer attack on new generation DRAMs to show the continuing threat of bit-flips. For the remaining attacks, we refer the readers to microarchitectural attacks survey [176].

3.4 Website Fingerprinting

In literature, the inference of visited websites has been investigated from many perspectives. Server-side website inference has been proposed through caching [46] and rendering [103] of website elements, visited URL styles [92], user interactions [165], quota management [96], and shared event loops [159]. Felten et al. [46] infer opened

websites from the server-side through caching, whereas Liang et al. [103] exploit the rendering of website elements and Vila et al. [159] use shared event loops. Vila et al. [159] use shared event loops to infer opened websites from the server side. Panchenko et al. [130] use traffic analysis to detect visited websites in the Tor network. Zhang et al. [178] exploit iOS APIs to infer visited websites and running applications. Spreitzer et al. [144, 145] obtain distinct features from the *procfs* filesystem and use Android APIs to infer opened web pages and applications. Lee et al. [101] exploit uninitialized GPU memory pages to detect websites, while Naghibijouybari et al. [124] exploit OpenGL APIs and GPU performance counters for this task. Gulmezoglu et al. [70] observe hardware performance events of modern processors to infer visited websites. Diao et al. [42] infer applications through system interrupts. Jana and Shmatikov [94] demonstrate that websites leave a distinct memory footprint in the browser application. Hornby [75] shows that this footprint can be observed from a malicious application via the processor cache. Oren et al. [127] as well as Gruss et al. [58] demonstrate that opened websites and their individual elements can be inferred from cache observations taken from a malicious JavaScript applet. Shusterman et al. [143] extend this work by inferring websites from JavaScript with simple last-level cache profiles that are classified by convolutional neural networks and long short-term memory.

3.5 Machine Learning and Side-Channel Attacks

Side-channel attacks (SCAs) typically rely on signal processing and statistics to infer information from observations. Since 2011, advanced machine learning approaches were introduced to side-channel literature. Lerman et al. [102] use random forests (RFs), SVMs, and self-organizing maps (SOMs) to compare the effectiveness of ma-

chine learning techniques against template attacks. Martinasek et al. [119, 118] showed that basic neural network techniques can recover AES keys with a 96% success rate. With the increasing popularity of deep learning, corresponding techniques were also studied for SCAs. In 2014, Zheng et al. [185] used CNNs to classify time series data with high accuracy. In 2015, Beltramelli [24] introduced Long-Short Term Memory (LSTM), a special DL technique, to classify 12 different keypads obtained from smart watch motion sensors. In 2016, Maghrebi et al. [113] compared four deep learning techniques with template attacks while attacking an unprotected AES implementation using power consumption. The results indicated that CNNs outperform template attacks thanks to their advanced feature extraction capability. In 2017, Schuster et al. [137] showed that encrypted streams can be used to classify videos with CNNs.

3.6 Defense Mechanisms against Attacks

Low-level performance monitoring events such as HPCs have been used as security sensors to detect malicious activities [115, 73]. Similar to [175, 168], *Numchecker* [163] and *Confirm* [164] adopt these sensors to detect control flow violations, which are applied to *rootkits* and *firmware modifications*, respectively. In addition, classical ML algorithms such as support vector machines (SVMs) and k-nearest neighbors (KNNs) are adapted to naive heuristic-based techniques for multi-class classification [22, 40]. The latter explores neural network in a supervised fashion [40]. Tang et al. [150] train One-Class Support Vector Machine (OC-SVM) with benign system behavior and detect the malware in the system.

Despite the detection of malware and rootkits in the system, HPCs have also been used to detect microarchitectural attacks. Firstly, Chiappetta et al. [35] pro-

poses to monitor HPCs and the data is analyzed by using Gaussian Sampling (GS) or probability density function (pdf) to detect the anomalies on cryptographic implementations dynamically. Later, Zhang et al. [177] apply Dynamic Time Wrapping (DTW) to catch cryptographic implementation executions in the victim VMs. Then, the number of cache misses and hits in the attacker VMs are monitored during the execution of sensitive operations. Briongos et al. [30] implement the Change Point Detection (CPD) technique to determine the sudden changes in the time series data to detect F+F, F+R, and P+P attacks. Finally, Mushtaq et al. [123] detect the cache oriented microarchitectural attacks with supervised Linear Discriminant Analysis (LDA), Support Vector Machine (SVM) and Linear Regression (LR) technique under various system loads. We further compare the most related works with *FortuneTeller* in Section 9.3.

Chapter 4

Co-location Detection

In this chapter, we analyze a new co-location detection technique as a first step of public cloud attacks. Typically, attackers own a VM in a public cloud to execute their attack codes in a shared server. However, it is not visible to attackers whether there are other VMs which are used by other clients. Mostly, co-located VMs' IPs are not known by the attackers which poses a challenge for an attacker to target a specific client. In addition, the old methods used in [134] such as ping timing, traffic rates and computational load variations are not applicable to public clouds anymore, which requires a new technique for co-location detection. In the following sections, a novel technique is introduced to detect the co-location with targeted victims in a public cloud environment.

4.1 Motivation

Cloud computing is getting more popular with the recent advances in computing research. Therefore, their security is also important to protect the costumers against potential microarchitectural attacks. These attacks violate the security and privacy of information. One of the examples is the cache attacks on ASLR [78] protection,

which enables the control-flow hijacking attacks in the cloud environments. There are also other techniques to exploit the vulnerable cryptographic implementations in the co-located VMs. While early works such as [181] still required attacker and victim to co-reside on the same core within a processor, latest works [85, 111] work across cores and managed even to drop the memory de-duplication requirement of Flush+Reload attacks [173, 171, 89, 67].

All of the above attacks rely on the attacker's ability to co-locate with a potential victim. While co-location is an immediate consequence of the benefits of cloud computing (better utilization of resources, lower cost through shared infrastructure etc.), whether *exploitable* co-location is possible or easy has so far not been studied in detail. In his seminal work, Ristenpart et al. [134] studied the general feasibility of co-location in Amazon EC2, the most popular public cloud service provider (CSP) then and now, in detail. However, the cloud landscape has changed significantly since then: The EC2 has grown exponentially and operates data centers around the globe. A myriad of competitors have popped up, all competing for the rapidly growing customer base [49]. CSPs are also more aware of the potential security vulnerabilities and have since worked on making their systems leak less information across VM boundaries. Furthermore, in their experiments, both co-located parties were colluding to achieve co-location. That is, both parties were willingly involved in communicating with the other to detect co-location. While being of high importance to show the feasibility in the first place, trying to co-locate with a specific and most likely unwilling target can be considerably harder. Since that initial work, until very recently only little work has dealt with a more detailed study on the difficulty of co-location. Therefore, we believe, the problem of co-location on cloud requires further in depth analysis examining different detection methods under diverse scenarios and access levels for the attacker.

4.2 Software Profiling on LLC

The software profiling method works in a realistic setting with minimal assumptions. The method focuses on a non-cooperative scenario where the victim is not involved in a covert communication and continues its regular operations. Memory de-duplication or any form of shared libraries is not required to implement software profiling based co-location detection. The attacker employs the Prime+Probe technique to monitor and profile a portion of the LLC while a targeted software is running. As for the memory addressing, we profile the targeted code address as a relative address to the page boundary. Since the targeted library is page aligned, target code's relative address (the page offset) remains the same between executions. Using this information, we reduce our search space in the detection stage as explained below.

The experiment server has 10 cores and 2048 LLC sets in total. While 11 bits (b_6 to b_{16}) are used to determine the cache set number, the first 6 bits (b_6 to b_{11}) are known to the attacker since virtual to physical address translation has no effect on the first 12 bits (b_0 to b_{11}). On the other hand, each core has its own LLC set which means there are 10 different slices for each cache set number. Therefore, we need to monitor 320 different set-slice pairs such as $X \mod 64 = Y$ where X is 320 different set numbers (since we have 10 cores and 32 different set numbers satisfying the equation) and Y is the first 6 bits of the set number for the desired function.

The targeted software implementation is the RSA decryption. In our scenario, the attacker sends a RSA decryption request to the victim and LLC sets are monitored by attacker in parallel. However, for the RSA detection, the slice-selection algorithm of the CPU is required to locate the targeted multiplication code in the LLC in a reasonable time. Without the algorithm, it would take too much time to

monitor potential cache sets. For our experiments, we have used the algorithm that was reverse engineered by Inci et. al in [81].

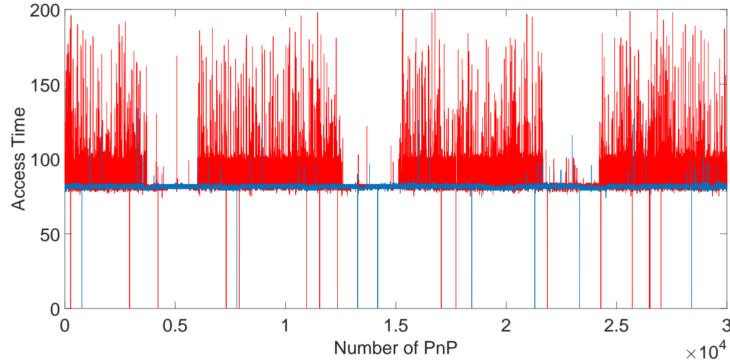
In summary, the co-location detection involves two steps for the software profiling on LLC;

- **Profiling Stage:** The first step of the profiling is to monitor the targeted LLC sets while the profiled code, the software is not running. After the regular operation of sets are observed, the RSA request will be sent to several candidate IP addresses. Before the detection stage is started all 320 set-slice pairs are profiled several times and the average access time to 20 (the set associativity) lines for each set-slice pair are calculated.

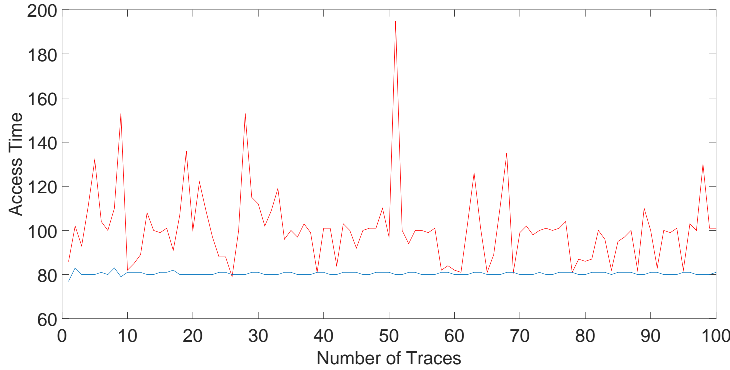
- **Detection Stage:** We send RSA decryption requests to candidate IPs in order to discover the IP address of the victim. After triggering the decryption we begin to monitor the portion of LLC to detect accesses due to the decryption. If we detect accesses in targeted set-slice pairs then we know that the correct IP address is found. As a double check, in addition to the RSA detection, we also detect AES encryption. Thus, we monitor another portion of the LLC where the AES T-tables potentially reside. If the victim is co-located with the attacker, we can detect and monitor these T-table accesses.

4.3 Results

We conducted the LLC Software Profiling experiments on the co-located Amazon EC2 instances with 10 core E5-2670 v2 processors. As for the software target, in order to demonstrate the versatility of the attack, we chose the RSA (Libgcrypt version 1.6.2) that uses sliding window exponentiation and the AES (OpenSSL version 1.0.1g, C implementation) that uses T-tables. Note that the detection method is



(a) RSA Pattern

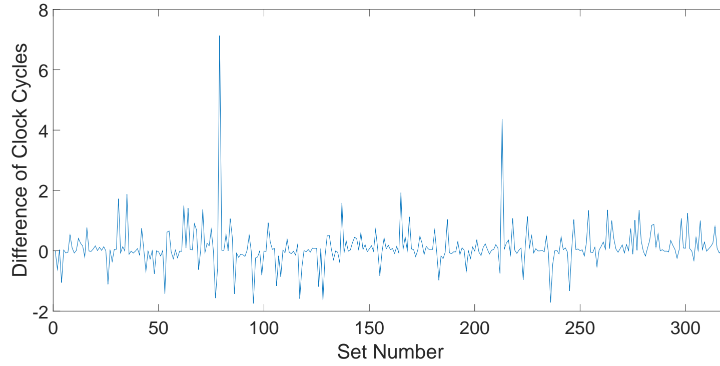


(b) AES Pattern

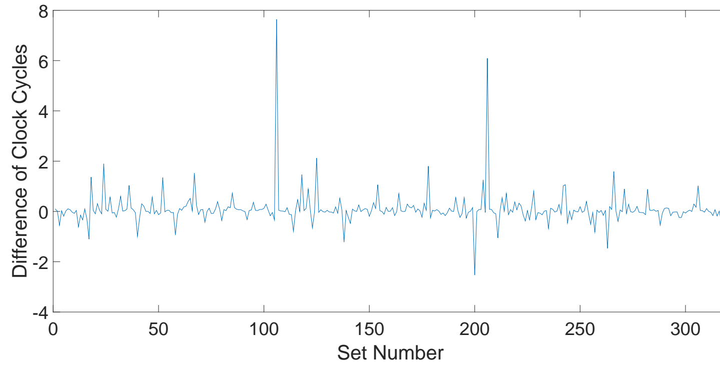
Figure 4.1: Red and blue lines represent idle and RSA decryption/AES encryption access times respectively

not limited to these targets since the attacker can run and profile any software in his instance and perform the attack.

For the RSA detection, the slice-selection algorithm of the CPU is required to locate the targeted multiplication code in the LLC within reasonable time. In our experiments, we have used the algorithm that was reverse engineered by Inci et. al in [81]. The first step of the profiling is to monitor the targeted LLC sets while the profiled code, RSA is not running. After the regular operation of sets are observed, the RSA request is sent to several IP addresses, starting from attacker's own subnet. As soon as the request is sent, the profiling starts and traces are recorded by the Prime+Probe. If the RSA decryption is running on the other VM, the pattern of



(a) RSA Analysis for the first co-located instance



(b) RSA Analysis for the second co-located instance

Figure 4.2: The difference of clock cycles between base and RSA decryption profiling for each set-slice pairs over 10 experiments

multiplication can be observed as in Figure 4.1. In general, the multiplication is performed between 2000-8000 traces. In these traces, we look for the delta of two profiles for each set-slice pair. In Figure 4.2, the difference between two profiles is illustrated for two co-located instances. Both figures show that there are two set-slice pairs with significantly higher access times (4-8 cycles) in average of 10 experiments. Hence, it can be concluded that these two sets are used by RSA decryption and this candidate instance is probably co-located with the attacker.

After we obtain IP addresses of several co-location candidates, we trigger an AES encryption by sending random ciphertexts and at the same time monitor the LLC. For this part of the detection stage, since AES encryption is much faster than RSA

decryption we can only catch one access to monitored T-table position. Hence, we send 100 AES encryption requests to each instance in the IP list. If we observe 90% cache miss for one of the set-slice pairs, it can be concluded that the AES encryption is performed by the co-located instance, as seen in Figure 4.1(b). After these steps, the attacker can start secret key recovery process in a public cloud.

4.4 Outcome

In conclusion, we represent three co-location detection methods that work in some of the most popular commercial clouds (Amazon EC2, Google Compute Engine, Microsoft Azure) and compare their efficiencies. In addition, for the first time we have achieved targeted co-locations in Amazon EC2 Cloud by applying the LLC software profiling for the AES encryption and the RSA decryption processes. For memory bus locking experiments, we have observed that if there is a frequent memory access, then there is more significant degradation. As for the cache covert channel, we showed that while it works in a cooperative scenario, it has high accuracy. And finally we presented the LLC software profiling technique that can be used variety of purposes including co-location detection without the help of memory de-duplication or cooperation from the victim side.

Chapter 5

Microarchitectural Attacks in the Cloud

After the co-location is established with a targeted victim, the attacker deploys a microarchitectural attack to leak sensitive information belonging to the victim. However, it is challenging to design various attack techniques which can work in public clouds. The main obstacle of the microarchitectural attacks is the need for high number of encryption/decryption request and measurements. To increase the performance of Flush+Reload attacks, a realistic Flush+Reload attack is introduced in the following section:

5.1 Faster Flush+Reload Attack

5.1.1 Motivation

Cloud computing and virtualization are getting popular more than ever with large companies like Microsoft, Google, Amazon, IBM, Oracle, Rackspace and many others investing billions of dollars trying to get a foothold in this new area of lucrative

business. This rapid increase in the number of cloud service providers is directly related to the emergence of server-less companies like Netflix, Dropbox, Instagram, Pinterest, Reddit, Imgur and many others that are using commercial cloud infrastructure [38]. Instead of buying expensive servers without knowing exactly how many of them they need, and then hiring IT personnel to maintain those servers, these fast growing companies have chosen to use public cloud systems to maintain their software and services.

The opportunities of using the commercials cloud are fairly obvious however, threats are not. Sharing a physical system between users reduces the cost while increasing the utilization hence the productivity. The isolation between the Virtual Machines (VM) in these systems is maintained by the Virtual Machine Manager (VMM) at the software level. However, software layer confinement techniques that force the *sandboxing* does not guarantee complete isolation and cannot ensure the prevention of data leakage from one VM to the other. While assigning only a single user to a physical machine, it goes against the idea of the cloud and prevents reaping of the benefits that come with the cloud systems. The most common source of information leakage across VM boundaries is the shared cache and the memory of the underlying physical system. Particularly memory deduplication allowed researchers to mount attacks that threaten both the user privacy and the security of the cryptographic systems.

In 2009, Ristenpart et al. [134] showed that it is possible to co-locate with a target on a cloud environment, namely Amazon EC2, and extract keystrokes from the co-located VM. In 2011, by exploiting the Kernel Samepage Merging (KSM), Suzaki et al. [149] was able to detect processes like `sshd`, `apache2`, `IE6` and `Firefox` running on a co-resident VM. The significance of this study is that it is possible to not just detect the existence of a target VM, but also detect running processes.

In 2013, Yarom et al. [173] applied the Flush+Reload attack across VMware VMs to recover a RSA key. Later in 2014, Irazoqui et al. [88] used Bernstein's AES cache timing attack to partially recover an AES key from various AES crypto library implementations in a cross-VM setting under XEN and VMware ESXI hypervisors. Also in 2014, Irazoqui et al. [89] implemented a cross-VM access driven cache attack on AES in a VMware ESXI system using the Flush+Reload attack.

Our Contribution

In this work, we implement for the first time a known-ciphertext cross-VM attack on AES using the Flush+Reload method and use three distinct data analysis methods to fully recover the secret key with varying encryption observations for different scenarios. For the attack, we take advantage of VMware ESXI' s memory deduplication mechanism called the Transparent Page Sharing. The attack is mounted on a multi-core high-end server, a specification found commonly on commercial cloud systems and does not require the attacker and the victim to be running on the same physical CPU core. Compared to the attack in [65], our attack is minimally invasive and works with less assumptions since the attacker does not need to control or exploit in any way the target process execution. Also compared to [89], the new attack does not assume to have access to the encryption server and works only by listening to the encryption server via cache covert channel and obtaining the ciphertexts from the network channel.

In summary, this work

- for the first time, mounts a cross-VM, **known-ciphertext only** AES key recovery attack using the Flush+Reload technique
- improves upon the previous cross-VM AES cache attacks by flushing **in be-**

tween the encryption rounds

- presents three distinct analysis methods that can be adapted to any table-based block ciphers

Our attack uses the side-channel technique known as Flush+Reload to monitor accesses to memory blocks. The Flush+Reload is applicable in the cross-VM setting if deduplication is enabled by the hypervisor and the monitored part of the memory is deduplicated. The latter is true if the monitored data is marked as shared (as is the common case for all crypto libraries) and the hypervisor has detected the duplicated data referenced from within both VMs. Also, different than the attack in [89], we utilize a separate AES detection step to detect the AES execution on the co-located target VM and eliminate the synchronization requirement with the server through the plaintext generation. This makes the proposed attack much more practical. We access the AES function memory address to detect the beginning of AES execution by Flush+Reload method. The reason why we access the memory location instead of simply running AES is that accessing a single memory location is much faster than running AES, allowing a higher attack resolution.

5.1.2 A single cache line attack on AES

The adversary monitors accesses to a single block of one of the T tables used in the last round of AES. In addition to the information t whether the T Table was accessed, the adversary needs to know the corresponding ciphertext c (or plaintext for a first-round attack). That is, we assume the adversary is able to collect several tuples $\langle c, t \rangle$. The monitored memory block corresponds to n T table entries $\mathbb{T}$ known to the adversary. For a monitored ciphertext byte C_i, these entries correspond to n T table outputs S_i, which are mapped one-to-one to n ciphertext byte values through

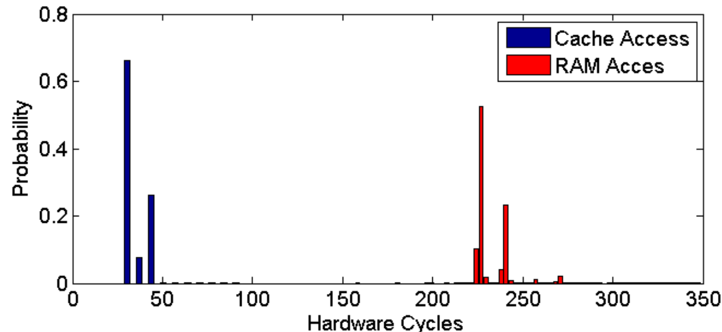
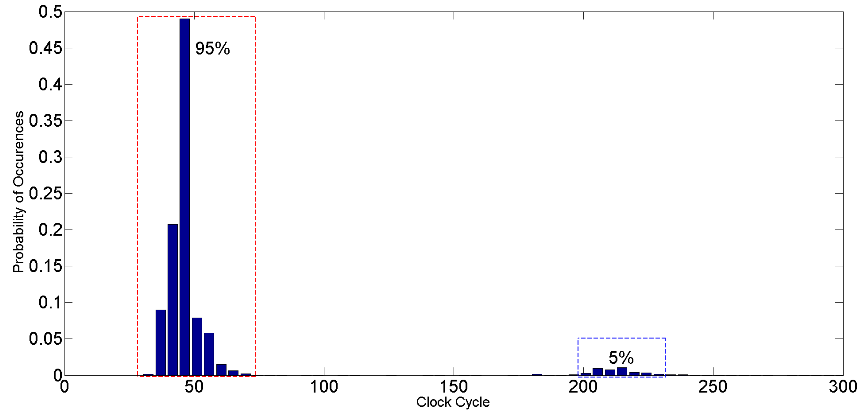


Figure 5.1: Data access time in hardware cycles when the data is located in the cache and in the memory

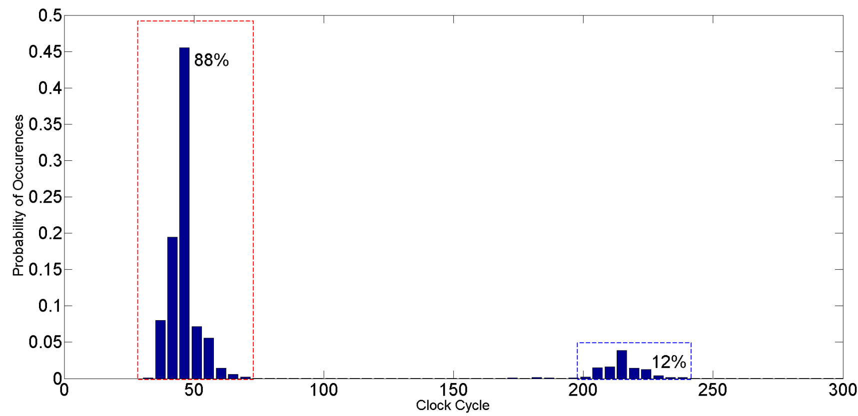
addition with the key. Hence, $c_{i,j} = k_i \oplus s_{i,j}$, where i is a byte position (ignoring the shift rows operation) and j indicates different values. If $s_{i,j}$ is equal to one of the values of the monitored T table memory block, i.e. $s_{i,j} \in \mathbb{T}$, then the monitored memory block will be accessed hence loaded to the cache. We will refer to this case as H_0. However, if $s_{i,j} \notin \mathbb{T}$, i.e. $s_{i,j}$ takes a value stored in a different memory block, then the monitored memory block is not loaded. Nevertheless, since each T table is accessed l times, there is still a high probability that the memory block was loaded by any of the other accesses. In fact, the probability that a memory block is not accessed during an encryption is given as: $\Pr[\text{no access to } T_j] = (1 - n/256)^l$. We will refer to this event as H_1.

For AES-128 in `OpenSSL 1.0.1g`, $n = 16$ and $l = 40$ per T_j, and therefore $100\% - \epsilon_0$ of reloads are expected to come from the cache in H_0, and only $92\% + \epsilon_1$ for H_1, where ϵ_i are noise terms. Hence, a side-channel containing information about memory/cache accesses will feature differing leakage distributions f_0 and f_1 for cases H_0 and H_1, respectively. To distinguish H_0 from H_1 the Flush+Reload method can be applied. In fact, using the Flush+Reload method, one can, with high probability, distinguish a cache access from a memory access as seen in Figure 5.1.

In our scenario (as described in Section 5.1.5) the leakage distributions f_0 and



(a) Distribution f_0 for case H_0



(b) Distribution f_1 for case H_1

Figure 5.2: Leakage Distributions f_0 and f_1 if Hypotheses H_0 and H_1 are correct. The measurements were taken in an Intel i5 2430M CPU in SSA scenario.

f_1 are depicted in Figure 5.2. The distributions are derived from the reload times measured by the Flush+Reload attack. The first peak in both distributions (at around 35 cycles) corresponds to a noisy cache reload, and the second peak (at around 220 cycles) corresponds to a memory reload. Since f_0 corresponds to H_0 and hence has more cache reloads than f_1, these distributions are distinguishable. This leakage was successfully exploited in [89].

5.1.3 Distinguishers for the AES Attack

To process the side-channel data, we describe and compare three distinguishers. The distinguishers we present here analyze one byte of the ciphertext c together with the access time t to the corresponding T table block to recover one byte k of the last round key.

As described earlier, our observations are split into two sets according to a hypothesis. If this hypothesis is correct, the resulting leakage distributions f_0 and f_1 for the two sets differ and hence, with sufficiently many observations—become distinguishable. For wrong key guesses, however, the hypotheses will be invalid, and both sets will sample from the same mixed distribution, making them indistinguishable. To detect whether samples for hypotheses H_0 and H_1 are actually from different distributions, we can apply several distinguishers. In the following we propose three distinguishers. The probably most common distinguisher is based on the difference of the means of the two distributions [98, 41]. As for the zero-value DPA [117], our hypothesis deviates from a single-bit prediction, yet, the test still just distinguishes two cases. Similarly, the variance test uses a statistical moment to distinguish the two distributions [99, 98, 41]. The last distinguisher applies a *miss counter*, as in [89]. The list is neither exhaustive, nor do we make an optimality claim. The latter is interesting future work that needs to be preceded by a better understanding and analysis of the underlying noise characterization, as noise can come from several different sources and is far from being Gaussian.

For the following descriptions we refer to the average miss counter value for H_i as $\overline{ctr}_{H_i}$, whereas we refer to the difference of means and difference of variances for f_i as $\overline{\tau}_{H_i}$ and $\operatorname{var}\tau_{H_i}$, respectively.

Miss-counter based Distinguisher

This distinguisher counts and compares the memory block misses for the two cases H_0 and H_1. Ideally, there should be no misses for H_0, as the memory block must have been accessed by the AES execution. To establish a *miss counter*, reload timings are converted to either a hit (0) or a miss (1), depending on whether the value is above or below a threshold access time. As seen in Figure 5.1, a good threshold for our processor and probing code is 130 cycles. Since H_1 contains significantly more values than H_0, we compare the relative counters instead of absolute ones. Our distinguisher becomes:

$$\mathcal{D}_{miss_ctr} = \arg\max_{\hat{k}} \left(\overline{ctr}_{H_1} - \overline{ctr}_{H_0}\right)$$

Difference of Means Distinguisher

The difference of means distinguisher approximates the means of the two distributions and outputs their difference in cycles.

$$\mathcal{D}_{means} = \arg\max_{\hat{k}} \left(\overline{\tau}_{H_1} - \overline{\tau}_{H_0}\right)$$

Since H_0 should feature more cache accesses than H_1, $\overline{\tau}_{H_0}$ is expected to be smaller, i.e. the biggest positive difference corresponds to the most likely key hypothesis. Welch's t-test distinguisher (which divides the means with their respective variance) can be equally well applied to guess the correct key. Indeed, Welch's t-test is commonly applied to check two hypothesis where two gaussian distributions have different means and variances. In this work, we studied Welch's t-test and did not obtain an improvement over the difference of means. Thus, we use the difference of means distinguisher due to its simplicity.

Variance based Distinguisher

The difference of variances distinguisher outputs the difference of variances in cycles.

$$\mathcal{D}_{vars} = \arg\max_{\hat{k}} (\text{var}\,\tau_{H_1} - \text{var}\,\tau_{H_0})$$

Note that, as before, the variance of H_0 should be smaller than that of H_1. However, outliers can badly affect this distinguisher. In cache attacks, significant outliers that can be orders of magnitude larger than regular data are not uncommon and need to be filtered to make this distinguisher work. Since H_i is key dependent, the guessed key $\hat{k}$ that maximizes the difference is the most likely to be correct. Note that the sign carries information in all three tests. In fact, the case H_0 and its leakage f_0 correspond to fewer cache misses, hence a lower miss counter, a lower average (mean) access time, and also a lower variance. The results will show that taking the sign into account derives a much better distinguisher.

When the three distinguishers are compared, the miss counter approach has the most interesting properties: It is quite intuitive, as cache misses and hits are what we are looking for. Furthermore, the method is only marginally affected by outliers. The main disadvantage of this method is the requirement of a threshold, which is processor-dependent and requires some minimal profiling. The other two methods are more affected by outliers. All three distinguishers can easily be converted to a correlation method. Indeed, by correlating the right term (e.g. $\overline{\tau}_{H_0}$) to 0 for H_0 (a guaranteed cache hit with low reload time) and 1 for H_1 (a possible cache miss with higher reload time), the most likely key $\hat{k}$ features the highest correlation.

5.1.4 Attack Scenarios

Next, we describe the principles of our new Flush+Reload attack as well as the original and the improved versions of the attack in [89]. We will refer to the attack in [89] as the Fully Synchronous Attack (FSA) and the improved version of it with the additional AES detection step as the Semi-Synchronous Attack (SSA). Finally, the attack scenario where the attacker requires no synchronization with the server will be referred as the Asynchronous Attack (ASA). In the following, we explain and compare these attacks in detail, listing challenges and advantages of each version.

FSA

This is the original attack used in [89] where the attacker first flushes the T tables, then sends a plaintext to the encryption server to trigger an AES encryption. The server receives the plaintext from the attacker, and sends the ciphertext back. Upon receipt of the ciphertext, the attacker reloads the monitored T table blocks to learn which entries were accessed by the encryption.

SSA

In this version of the attack, we improved over the FSA by detecting the AES encryption using the Flush+Reload method but there is still a need for trigger event by the adversary. The advantage of this attack is the usage of an AES encryption detector that detects whether the victim is performing an AES encryption. Once the AES encryption function call is detected, the attacker flushes the monitored T table blocks **during** the AES execution in between AES rounds. Flushing in between rounds reduces the number $l-1$ of unrelated accesses to the T table accesses, hence increasing the number of memory accesses for case H_1. In addition, we know that the detection algorithm takes half of the timing of an AES encryption. Therefore,

at least half of the rounds of the AES encryption is eliminated by this detection mechanism. This results in a more biased distribution f_1, i.e. a stronger leakage. Consequently, the attack succeeds with fewer encryptions.

ASA

In the ASA, we improve over the previous two attacks by not requiring any trigger event by the adversary. Instead, plaintexts are generated by the server in regular intervals of 5M cycles. The adversary uses an AES detector to detect the AES function call and perform the Flush+Reload attack on the fly. In addition the network is monitored to recover transmitted ciphertexts. Unlike the previous attacks, this attack is a true ciphertext-only attack.

Note that the ASA presents a more realistic attack scenario than those presented in [65] and [89]. In [65] Gullasch et al. described a Flush+Reload attack on AES implementation of the `OpenSSL` library where they overload the CPU and suspend the AES encryption by controlling CFS. In [89] authors require synchronization with the server through the plaintext generation. In contrast to these previous attacks, our attack differs in the following ways:

- Our attack flushes the T tables **during** the AES encryption rather than before;
- CFS exploitation or any other type of CPU overloading is not necessary;
- Synchronization through the plaintext is no longer required, but the AES encryption call is detected instead;
- Improved side-channel data analysis/key recovery methods recover the key with fewer encryptions.

5.1.5 Experiment Setup

For the experiments, we have used the following two setups;

- **Native Execution:** In this setup, the AES encryption process and the attacker run on a native Ubuntu 12.04 LTS version with no virtualization. In this setting, we have used a two core Intel i5-2430M CPU clocked at 2.4 GHz. The purpose of this scenario is to run the attack in an environment with minimal noise and to achieve comparability to former non cross-VM cache attacks.

- **Cross-VM Execution:** In this setup, two up-to-date Ubuntu VMs, VM1 and VM2 are launched and managed by VMware ESXI 5.5 baremetal hypervisor. The attacks are then performed across hypervisor isolation boundaries. The first VM is used as the target that does the AES encryption while the second VM acts as the attacker and executes the Flush+Reload attack, trying to recover the secret key. The experiments in this setting were performed on an Intel Xeon E5-2670 v2 CPU. This setup reflects a realistic attack scenario by using a modern CPU commonly used in commercial cloud systems [1, 5]. In this setup, data access from the cache takes 30 cycles and the memory takes 233 cycles on average. Also in the same specification, single AES encryption without and with pre-flushed T-tables requires 257 and 659 cycles, respectively. As the Figure 5.1 shows, the timing separation between the CPU cache and the main memory is clear with very few outliers. We further observe in Figure 5.1 that the AES execution time changes greatly depending on whether or not the T-tables used for the encryption are loaded in the cache.

Note that all the timing measurements in the experiments are gathered using the `Read Time Stamp Counter and Processor ID (RDTSCP)` instruction. The usage

of the `RDTSCP` instruction is allowed in VMware user mode, but not in KVM. Moreover, this instruction is not emulated by the VMM but executed directly, unlike other serializing instructions like `CPUID` used in [89]. Also, the flushing operation is performed using the `Cache Line Flush (CLFLUSH)` instruction.

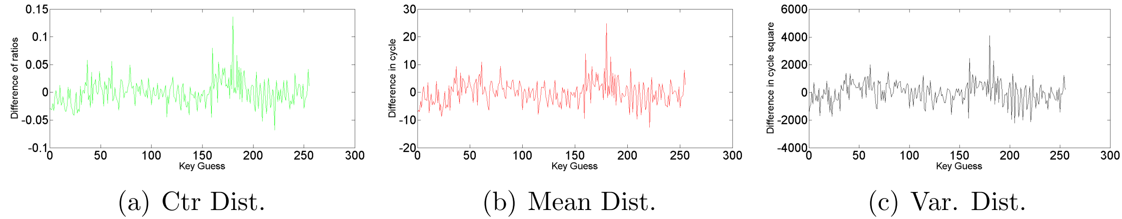
In all experiments, one target process executes AES encryption while the attacker process tries to recover the secret key by monitoring the T-tables with the Flush+Reload technique. In order to clearly show the the attack success under different assumptions, we have used two distinct attack environments.

5.1.6 Results

We performed the experiments for all three attack scenarios, i.e. FSA, SSA and ASA in both native and virtualized environments. Furthermore, we analyze the timing behavior to show the improvement on the success rate by using the three different distinguishers mentioned in Section 5.1.3: the miss counter distinguisher, the difference of means distinguisher and the difference of variances distinguisher.

At first we present and compare the scores of the key guesses using the three different distinguishers in native execution in Figure 5.3. The difference of means and variances distinguishers suffer more from noise due to heavy outliers stemming from different microarchitectural sources of noise. However the experiments shown in Figure 5.3 were taken cutting off outliers with an outlier threshold value of 5 times the memory access time. It can be seen that for 10,000 encryptions the three distinguishers clearly maximize the score for the correct key, i.e. 180 in this case.

Then, the results of the three different attack scenarios is presented in Table 5.1 by comparing the ratio between cache accesses and memory accesses for cases H_0 and H_1. The precise distribution for the SSA scenario was given in Figure 5.2. Recall that without noise, the ratio should be 100%/0% for H_0 vs. 92%/8% for H_1



(a) Ctr Dist. (b) Mean Dist. (c) Var. Dist.

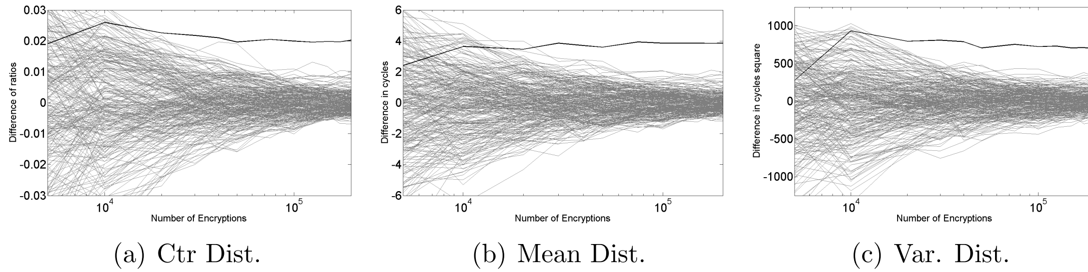
Figure 5.3: Comparison of the scores of key guesses in the natively executed FSA scenario for three different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c), applied to 10000 traces. The correct key is 180 and clearly distinguishable in all three cases.

for the FSA scenario and even more biased for the SSA scenario. The probability distribution shows that for H_0 approximately 95% of the reload values are coming from L3 cache while only the 5% come from the main memory. In H_1 however, the reload values coming from L3 cache are down to 88%, while the values coming from the main memory increase to 12%. Also, it can be seen from the Table 5.1 that there is a significant improvement in the distinguishability for SSA scenario due to flushing during AES encryption. Flushing during the encryption translates into lower noise in the T table measured access times and an improved success rate. However, the increased number of detected memory accesses for SSA is likely caused by flushes occurring *after* AES encryption has terminated. Thus, although the more realistic ASA scenario decreases the success rate, in comparison to the SSA scenario due to the difficulty of the AES detection. Hence, SSA is the most efficient way to decrease the noise and have a good resolution to find the correct key with a small number of encryptions.

Finally the number of traces needed for the recovery of the key are presented in Figures 5.4,5.5. As for the attack scenario success rates, our experiments in the native execution setting show that the SSA yields higher success rate than the FSA and the ASA which require 3,000, 25,000 and 30,000 encryptions ,respectively. Also, the variance distinguisher works better in native setting than the other two

Table 5.1: Distribution of cache accesses vs. memory accesses for the two hypotheses over the three attack scenarios. SSA provides the best distinguishability.

Attack Scenarios	$\mathbf{H}_0$		$\mathbf{H}_1$	
	Cache	Memory	Cache	Memory
Ideal case	100%	0%	92%	8%
FSA	99%	1%	97%	3%
SSA	95%	5%	88%	12%
ASA	97%	3%	96%	4%



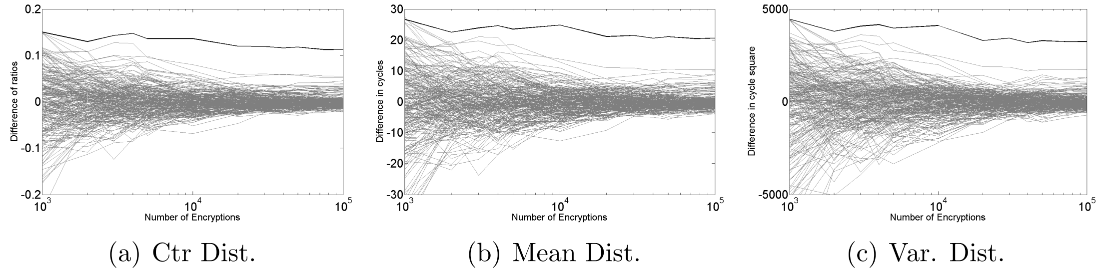
(a) Ctr Dist. (b) Mean Dist. (c) Var. Dist.

Figure 5.4: Comparison of results in native execution for FSA scenario for different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c).

distinguishers. For other attack scenarios e.g. the ASA, the mean distinguisher works the best, see Figure 5.5(b). Note that, since ASA is the most realistic scenario, it requires more encryption samples than the other two, most notably compared to the SSA where only 3,000 encryption samples are needed.

5.1.6.1 Cross-VM Execution Results

In the cross-VM setting, the FSA scenario requires 30,000 encryptions to recover the full key using the miss counter hypothesis as seen in Figure 5.6(a). In the same setting, 50,000 encryptions are needed when the difference of means distinguisher is used as in Figure 5.6(d). As for the SSA, only 10,000 encryptions are enough to recover the full key using the mean distinguisher in Figure 5.6(b). If the miss counter distinguisher is used instead of the mean distinguisher, 40,000 encryptions



(a) Ctr Dist. (b) Mean Dist. (c) Var. Dist.

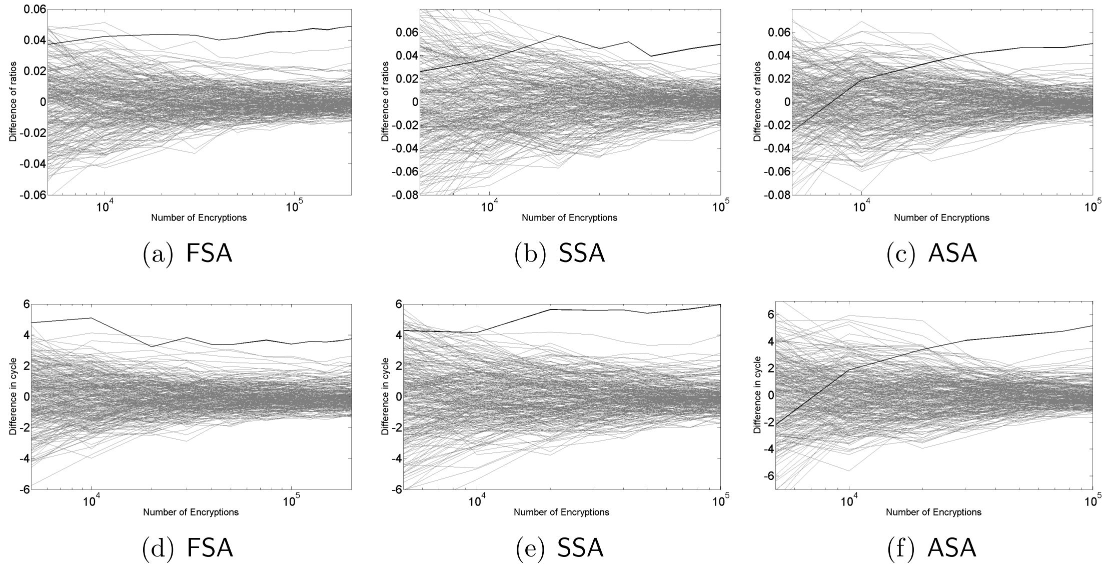
Figure 5.5: Comparison of results in native execution for the SSA for different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c).

are needed as seen in Figure 5.6(e).

For the ASA scenario, 30,000 encryptions are enough to recover the full key using miss-counter and mean distinguishers as seen in Figures 5.6(c), 5.6(f). Also, when we compare different distinguisher methods in the cross-VM setting for different attack scenarios, we see that the difference of means distinguisher works better than the miss-counter distinguisher for the most successful attack which is the SSA. While the miss-counter distinguisher gives better results for the FSA, the two distinguishers have the same impact on the results for the ASA which is the most realistic attack scenario.

We would like to note that the difference of means and the difference of variances distinguishers work better in the SSA and ASA scenarios, whereas the miss counter yields better results for the FSA. Moreover, the main advantage of using the variance and mean distinguishers is that they do not need an architecture dependent threshold, whereas the miss counter approach needs the access time distribution of the cache hierarchy.

Also note that the improvement of SSA is due to flushing **during** the AES execution which yields lower noise in the reloading stage. As for the ASA, we would like to emphasize that the higher number of encryptions requirement is due to the



(a) FSA (b) SSA (c) ASA

(d) FSA (e) SSA (f) ASA

Figure 5.6: Results in cross-VM execution for different attack scenarios using the miss counter distinguisher FSA (a) SSA (b) ASA (c) and the means distinguisher FSA (d) SSA (e) and ASA (f).

more realistic nature of the attack setting i.e. the lack of synchronization between the server and the spy process. Finally, we would like to remark that **only 15 seconds** are enough to recover the whole key in SSA scenario, which to the best of our knowledge is the fastest working attack in a realistic cross-VM setting without the scheduler exploitation.

5.1.7 Outcome

In conclusion, for the first time we have accomplished a cache side-channel attack on AES by flushing in between rounds. We also used an additional AES detection stage to create an asynchronous attack setting. In addition to that, we improved upon the previous work on cross-VM AES attacks by utilizing three different distinguishers for the key recovery. Finally, our experiments show that among three attack scenarios, SSA works with the minimum number of encryptions, requiring only 3,000 in the native and 10,000 in the cross-VM setting.

5.2 Prime+Probe Attack on Amazon Cloud

5.2.1 Motivation

Cloud computing services are more popular than ever with their ease of access, low cost and real-time scalability. With increasing adoption of cloud, concerns over cloud specific attacks have been rising and so has the number of research studies exploring potential security risks in the cloud domain. A main enabler for cloud security is the seminal work of Ristenpart et al. [134]. The work demonstrated the possibility of co-location as well as the security risks that come with it. The co-location is the result of resource sharing between tenant Virtual Machines (VMs). Under certain conditions, the same mechanism can also be exploited to extract sensitive information from a co-located victim VM, resulting in security and privacy breaches. Methods to extract information from VMs have been intensely studied in the last few years however remain infeasible within public cloud environments, e.g. [184, 126, 148]. The potential impact of attacks on crypto processes can be even more severe, since cryptography is at the core of any security solution. Consequently, extracting cryptographic keys across VM boundaries has also received considerable attention lately. Initial studies explored the Prime+Probe technique on L1 cache [182, 65]. Though requiring the attacker and the victim to run on the same physical CPU core simultaneously, the small number of cache sets and the simple addressing scheme made the L1 cache a popular target. Follow up works have step by step removed restrictions and increased the viability of the attacks. The shared Last Level Cache (LLC) now enables true cross-core attacks [172, 25, 89] where the attacker and the victim share the CPU, but not necessarily the CPU core. Most recent LLC Prime+Probe attacks no longer rely on de-duplication [111, 85] or core sharing, making them more widely applicable.

With the increasing sophistication of attacks, participants of the cloud industry ranging from Cloud Service Providers (CSPs), to hypervisor vendors, up all the way to providers of crypto libraries have fixed many of the newly exploitable security holes through patches [3, 11, 9]—many in response to published attacks. However, many of the outdated cryptographic libraries are still in use, opening the door for exploits. A scan over the entire range of IPs in the South America East region yields that 55% of TLS hosts installed on Amazon EC2 servers have not been updated since 2015 and are vulnerable to an array of more recently discovered attacks. Consequently, a potential attacker such as a nation state, hacker group or a government organization can exploit these vulnerabilities for bulk recovery of private keys. Besides the usual standard attacks that target individuals, this enables mass surveillance on a population thereby stripping the network from any level of privacy. Note that the attack is enabled by our trust in the cloud. The cloud infrastructure already stores the bulk of our sensitive data. Specifically, when an attacker instantiates multiple instances in a targeted availability zone of a cloud, she co-locates with many vulnerable servers. In particular, an attacker trying to recover RSA keys can monitor the LLC in each of these instances until the pattern expected by the exploited hardware level leakage is observed. Then the attacker can easily scan the cloud network to build a public key database and deduce who the recovered private key belongs to. In a similar approach, Heninger et al. [72] scan the network for public keys with shared or similar RSA modulus factors due to poor randomization. Similarly Bernstein et al. [27] compiled a public key database and scanned for shared factors in RSA modulus commonly caused by broken random number generators.

In this work, we explore the viability of full RSA key recovery in the Amazon EC2 cloud. More precisely, we utilize the LLC as a covert channel both to co-locate and perform a cross-core side-channel attack against a recent cryptographic implementa-

tion. Our results demonstrate that even with complex and resilient infrastructures, and with properly configured random number generators, cache attacks are a big threat in commercial clouds.

Our Contribution

This work presents a full key recovery attack on a modern implementation of RSA in a commercial cloud and explores all steps necessary to successfully recover both the key and the identity of the victim. This attack can be applied under two different scenarios:

1. **Targeted Co-location:** In this scenario, we launch instances until we co-locate with the victim as described in [158, 82]. Upon co-location the secret is recovered by a cache enabled cross-VM attack.

2. **Bulk Key Recovery:** We randomly create instances and using cross-VM cache attacks recover imperfect private keys. These keys are subsequently checked and against public keys in public key database. The second step allows us to eliminate noise in the private keys and determine the identity of the owner of the recovered key.

Unlike in earlier bulk key recovery attacks [72, 27] we do not rely on faulty random number generators but instead exploit hardware level leakages.

Our specific technical contributions are as follows:

- We first demonstrate that the LLC contention based co-location detection tools are plausible in public clouds

- Second, we reverse-engineer the undocumented non-linear slice selection algorithm implemented in Intel Xeon E5-2670 v2 [6] used by our Amazon EC2

instances, and utilize it to automate and accelerate the attack

- Third, we describe how to apply the Prime+Probe attack to the LLC and obtain RSA leakage information from co-located VMs

- Last, we present a detailed analysis of the necessary post-processing steps to cope with the noise observed in a real public cloud setup, along with a detailed analysis on the CPU time (at most 30 core-hours) to recover both the noise-free key and the owner's identity (IP).

5.2.2 Cross-VM RSA Key Recovery

To prove the viability of the Prime+Probe attack in Amazon EC2 across co-located VMs, we present an expanded version of the attack implemented in [111] by showing its application to RSA. It is important to remark that the attack *is not* processor specific, and can be implemented in any processor with inclusive last level caches. In order to perform the attack:

- We make use of the fact that the offset of the address of each table position entry does not change when a new decryption process is executed. Therefore, we only need to monitor a subsection of all possible sets, yielding a lower number of traces.

- Instead of the monitoring both the multiplication and the table entry set (as in [111] for El-Gamal), we *only monitor a table entry set in one slice.* This avoids the step where the attacker has to locate the multiplication set and avoids an additional source of noise.

The attack targets a sliding window implementation of RSA-2048 where each position of the pre-computed table will be recovered. We will use Libgcrypt 1.6.2 as

our target library, which not only uses a sliding window implementation but also uses CRT and message blinding techniques [104]. The message blinding process is performed as a side channel countermeasure for `chosen-ciphertext` attacks, in response to studies such as [52, 50].

Algorithm 1 RSA with CRT and Message Blinding

Input: $c \in \mathbb{Z}_N$, Exponents d, e, Modulus $N = pq$
Output: m
$r \xleftarrow{\$} Z_N$ with $\gcd(r, N) = 1$ ▷ Message Blinding
$c^* = c \cdot r^e \mod N$
$d_p = d \mod (p-1)$ ▷ CRT conversion
$d_q = d \mod (q-1)$
$m_1 = (c^*)^{d_p} \mod p$ ▷ Modular Exponentiation
$m_2 = (c^*)^{d_q} \mod q$
$h = q^{-1} \cdot (m_1 - m_2) \mod p$ ▷ Undo CRT
$m^* = m_2 + h \cdot q$
$m = m^* \cdot r^{-1} \mod N$ ▷ Undo Blinding
return m

We use the Prime+Probe side channel technique to recover the positions of the table T that holds the values $c^3, c^5, c^7, \ldots, c^{2^W - 1}$ where W is the window size. For CRT-RSA with 2048 bit keys, $W = 5$ for both exponentiations d_p, d_q. Observe that, if all the positions are recovered correctly, reconstructing the key is a straightforward step.

Recall that we do not control the victim's user address space. This means that we do not know the location of each of the table entries, which indeed changes from execution to execution. Therefore we will monitor a set hoping that it will be accessed by the algorithm. However, our analysis shows a special behavior: each time a new decryption process is started, even if the location changes, the offset field does not change from decryption to decryption. Thus, we can *directly* relate a monitored set with a specific entry in the multiplication table.

The knowledge of the processor in which the attack is going to be carried out

gives an estimation of the probability that the set/slice we monitor collides with the set/slice the victim is using. For each table entry, we fix a specific set/slice where not much noise is observed. In the Intel Xeon E5-2670 v2 processors, the LLC is divided in 2048 sets and 10 slices. Therefore, knowing the lowest 12 bits of the table locations, we will need to monitor *one* set/slice that solves $s \mod 64 = o$, where s is the set number and o is the offset for a table location. This increases the probability of probing the correct set from $1/(2048 \cdot 10) = 1/20480$ to $1/((2048 \cdot 10)/64) = 1/320$, reducing the number of traces to recover the key by a factor of 64. Thus our spy process will monitor accesses to *one* of the 320 set/slices related to a table entry, hoping that the RSA encryption accesses it when we run repeated decryptions. Thanks to the knowledge of the non linear slice selection algorithm, we can easily change our monitored set/slice if we see a high amount of noise in one particular set/slice. Since we also have to monitor a different set per table entry, it also helps us to change our eviction set accordingly. The threshold is different for each of the sets, since the time to access different slices usually varies. Thus, the threshold for each of the sets has to be calculated before the monitoring phase. In order to improve the applicability of the attack the LLC can be monitored to detect whether there are RSA decryptions or not in the co-located VMs as proposed in [82]. After it is proven that there are RSA decryptions the attack can be performed.

In order to obtain high quality timing leakage, we synchronize the spy process and the RSA decryption by initiating a communication between the victim and attacker, e.g. by sending a TLS request. Note that we are looking for a particular pattern observed for the RSA table entry multiplications, and therefore processes scheduled before the RSA decryption will not be counted as valid traces. In short, the attacker will communicate with the victim before the decryption. After this initial communication, the victim will start the decryption while the attacker starts

monitoring the cache usage. In this way, we monitor 4,000 RSA decryptions with the same key and same ciphertext for each of the 16 different sets related to the 16 table entries.

We investigate a hypothetical case where a system with dual CPU sockets is used. In such a system, depending on the hypervisor CPU management, two scenarios can play out; processes moving between sockets and processes assigned to specific CPUs. In the former scenario, we can observe the necessary number of decryption samples simply by waiting over a longer period of time. In this scenario, the attacker would collect traces and only use the information obtained during the times the attacker and the victim share sockets and discard the rest as missed traces. In the latter scenario, once the attacker achieves co-location, as we have in Amazon EC2, the attacker will always run on the same CPU as the target hence the attack will succeed in a shorter span of time.

5.2.3 Leakage Analysis Method

Once the online phase of the attack has been performed, we proceed to analyze the leakage observed. There are three main steps to process the obtained data. The first step is to identify the traces that contain information about the key. Then we need to synchronize and correct the misalignment observed in the chosen traces. The last step is to eliminate the noise and combine different graphs to recover the usage of the multiplication entries. Among the 4,000 observations for each monitored set, only a small portion contains information about the multiplication operations with the corresponding table entry. These are recognized because their exponentiation trace pattern differs from that of unrelated sets. In order to identify where each exponentiation occurs, we inspected 100 traces and created the timeline shown in Figure 5.7(b). It can be observed that the first exponentiation starts after 37% of

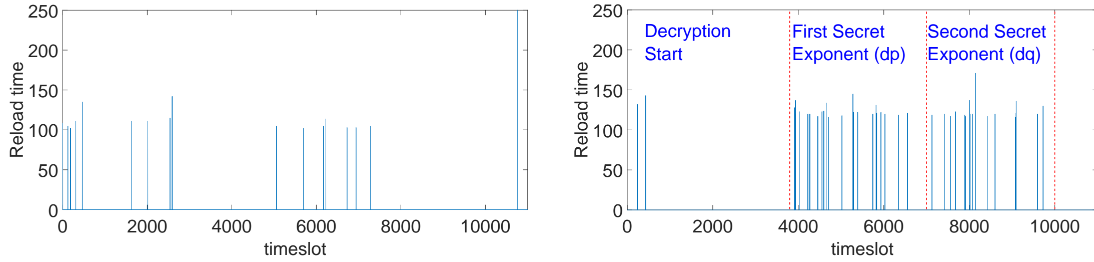


Figure 5.7: Different sets of data where we find a) trace that does not contain information b) trace that contains information about the key

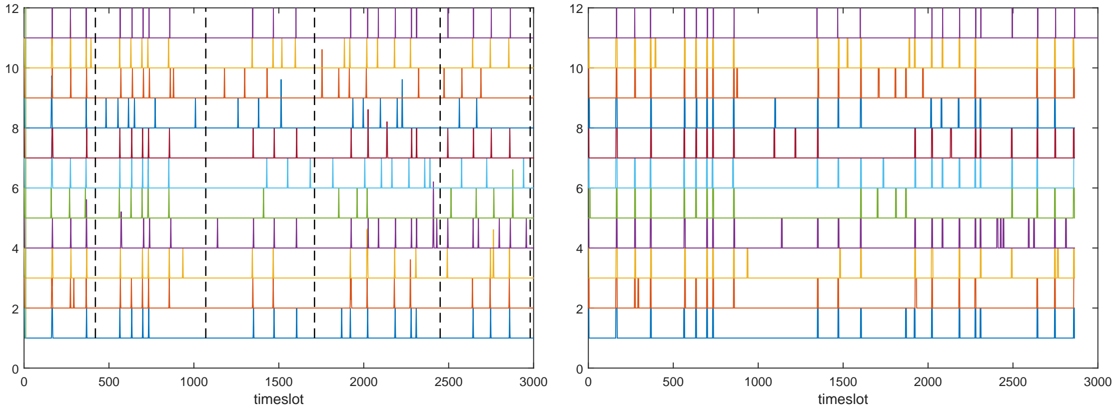


Figure 5.8: 10 traces from the same set where a) they are divided into blocks for a correlation alignment process b) they have been aligned and the peaks can be extracted

the overall decryption time. Note that among all the traces recovered, only those that have more than 20 and less than 100 peaks are considered. The remaining ones are discarded as noise. Figure 5.7 shows measurements where no correct pattern was detected (Fig. 5.7(a)), and where a correct pattern was measured (Fig. 5.7(b)).

In general, after the elimination step, there are 8−12 correct traces left per set. We observe that data obtained from each of these sets corresponds to 2 consecutive table positions. This is a direct result of CPU cache prefetching. When a cache line that holds a table position is loaded into the cache, the neighboring table position is also loaded due to cache locality principle.

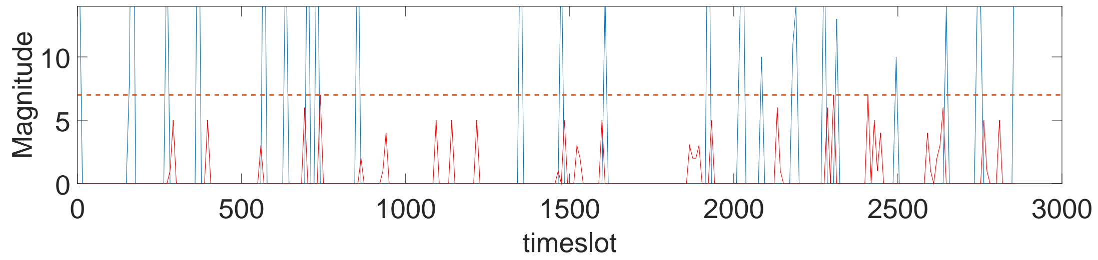


Figure 5.9: Eliminating false detections using a threshold (red dashed line) on the combined detection graph.

For each graph to be processed, we first need to align the creation of the look-up table with the traces. Identifying the table creation step is trivial since each table position is used twice, taking two or more time slots. Figure 5.8(a) shows the table access position indexes aligned with the table creation. In the figure, the top graph shows the true table accesses while the rest of the graphs show the measured data. It can be observed that the measured traces suffer from misalignment due to noise from various sources e.g. RSA or co-located neighbors.

To fix the misalignment, we take most common peaks as reference and apply a correlation step. To increase the efficiency, the graphs are divided into blocks and processed separately as seen in Figure 5.8(a). At the same time, Gaussian filtering is applied to peaks. In our filter, the variance of the distribution is 1 and the mean is aligned to the peak position. Then for each block, the cross-correlation is calculated with respect to the most common hit graph i.e. the intersection set of all graphs. After that, all graphs are shifted to the position where they have the highest correlation and aligned with each other. After the cross-correlation calculation and the alignment, the common patterns are observable as in Figure 5.8(b). Observe that the alignment step successfully aligns measured graphs with the true access graph at the top, leaving only the combining and the noise removal steps. We combine the graphs by simple averaging and obtain a single combined graph.

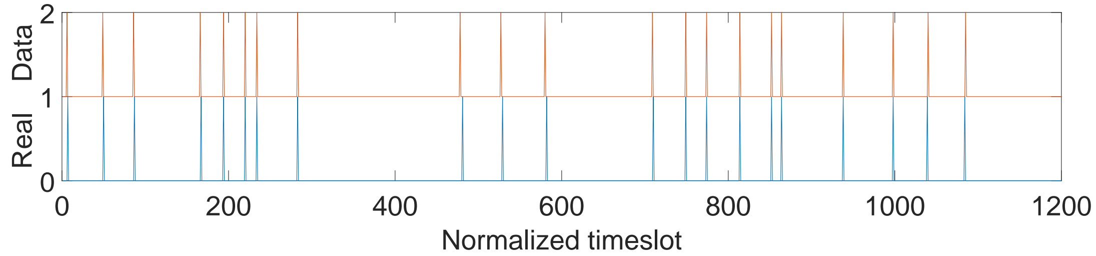


Figure 5.10: Comparison of the final obtained peaks with the correct peaks with adjusted timeslot resolution

In order to get rid of the noise in the combined graph, we applied a threshold filter as can be seen in Figure 5.9. We used 35% of the maximum peak value observed in graphs as the threshold value. Note that a simple threshold was sufficient to remove noise terms since they are not common between graphs.

Now we convert scaled time slots of the filtered graph to real time slot indexes. We do so by dividing them with the spy process resolution ratio, obtaining the Figure 5.10. In the figure, the top and the bottom graphs represent the true access indexes and the measured graph, respectively. Also, note that even if additional noise peaks are observed in the obtained graph, it is very unlikely that two graphs monitoring consecutive table positions have noise peaks at the same time slot. Therefore, we can filter out the noise stemming from the prefetching by combining two graphs that belong to consecutive table positions. Thus, the resulting indexes are the corresponding timing slots for look-up table positions.

The very last step of the leakage analysis is finding the intersections of two graphs that monitor consecutive sets. By doing so, we obtain accesses to a single table position as seen in Figure 5.11 with high accuracy. At the same time, we have total of three positions in two graphs. Therefore, we also get the positions of the neighbors. A summary of the result of the leakage analysis is presented in Table 5.2. We observe that more than 92% of the recovered peaks are in the correct position.

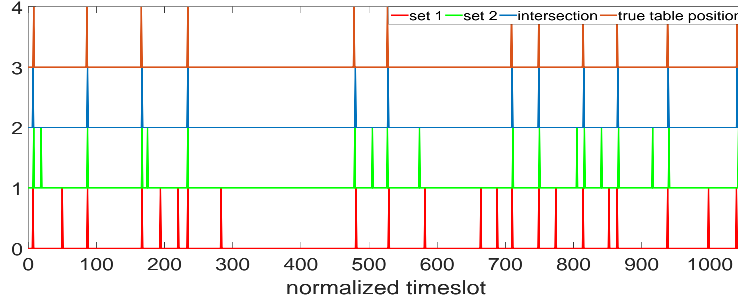


Figure 5.11: Combination of two sets

Table 5.2: Successfully recovered peaks on average in an exponentiation

Average Number of traces/set	4000
Average number of correct graphs/set	10
Wrong detected peaks	7.19%
Missdetected peaks	0.65%
Correctly detected peaks	92.15%

However, note that by combining two different sets, the wrong peaks will disappear with high probability, since the chance of having wrong peaks in the same time slot in two different sets is very low.

5.2.4 Outcome

In conclusion, we show that even with advanced isolation techniques, resource sharing still poses security risk to public cloud customers that do not follow the best security practices. The cross-VM leakage is present in public clouds and can be a practical attack vector for data theft. Therefore, users have a responsibility to use latest improved software for their critical cryptographic operations. Even further, we believe that smarter cache management policies are needed both at the hardware and software levels to prevent side-channel leakages.

Chapter 6

Machine Learning based Application Detection in the Cloud

6.1 Motivation

In the last decade the cloud infrastructure has matured to the point where companies, government agencies, hospitals and schools alike have outsourced their infrastructure to cloud service providers. The main benefit of moving to the cloud is the reduction of money spent on IT by pooling servers and storages in bigger cloud services. In many cases, rented servers are instances shared through virtualization among many users. Sharing is the basis for the reduction in the IT costs. Despite the clear cost benefit, given the vast amount of personal and sensitive information kept on shared resources, rightfully security concerns have been steadily growing among cloud customers.

Naturally, the cloud infrastructure has come under the scrutiny of security re-

searchers. The first breakthrough result was reported by Ristenpart et al. [134] who showed that it is possible to co-locate in a controlled manner with possible target instances on commercial public clouds, e.g. Amazon EC2. This work opened the door to a series of investigations that examined the threats from an attacker, i.e. a legitimate cloud user, exploiting cross VM leakage to steal sensitive information. A number of methods have been proposed to steal private keys or valuable information between VMs in IaaS and PaaS clouds [89, 81, 68, 181, 183]. In these works, the cryptographic keys and other sensitive information are stolen by attacker by exploiting leakages at the microarchitectural level, i.e. through the shared cache architecture. Especially, the shared last-level cache (LLC) is a dangerous information leakage source in public clouds. IaaS instance allocation policy commonly allocate an instance per core. This means that the LLC is a shared resource among multiple user instances. Thus by timing his own LLC access times, a user can gleam information on another user's co-located instance's cache access behavior. LLC attacks proliferated to the point that the most recent LLC Prime&Probe attacks do not depend on the de-duplication feature [85, 111] to be enabled to mount cross core cache attacks in public commercial clouds.

Cross-VM leakage attacks are extremely destructive in nature. They require almost no privileges. Anyone can rent an instance on EC2 for a small fee and run an attack code on an instance co-located typically with multiple target instances out of potentially millions of targets. The attack code does only legitimate accesses, e.g. collection of cache access times, for accesses in its own memory/application space. Thus, Cross-VM attacks pose a great threat. Potentially, one could automate the attack and mine the entire compute cloud for cryptographic keys. There are practical difficulties in carrying out such attacks on a mass scale. Cross-VM security attacks on public clouds require a sophisticated methodology to extract the

sensitive information from data. For instance, the cache pattern is extracted and by using personal effort the relation between pattern and key is established [81]. This makes discovery of vulnerabilities, a manual process, rather costly and time-consuming. Cryptographic library designers experience a similar difficulty. Cryptographic libraries are constantly patched for newly discovered leakages and emerging vulnerabilities. This in itself is a painstaking process requiring careful inspection of the code for any potential leakage for a target platform[1]. Software bugs may result in secondary leakages confounding the problem further. Thus, in practice, even constant execution flow/time implementation may be compromised due to bugs.

With the growing complexity of cryptographic libraries, or more broadly of code that handles sensitive data, it becomes impossible to manually verify the code for leakages across the numerous platforms exhaustively. Clearly, there is a great need for automated verification and testing of sensitive code against leakages. Firstly, Brumley et al. [31] proposed vector quantization and HMM to classify ECC ops with respect to L1-D cache. Then, an automated profiling attack on LLC was introduced by Gruss et al. [63]. In this work, the access pattern of different events are first extracted in a non-virtualized (less noisy) environment. The attacker learns the cache access templates from the cache. During an attack the new data is compared against the learned templates. While this is a worthy effort, machine learning (ML) algorithms have advanced to the point where they offer sophisticated solutions to complicated recognition, classification, clustering, and regression problems. For instance, image and speech recognition, sense extraction in text and speech [129], recommendation systems and search engines, as well as malicious behavior detection [33]. Further, cryptographers recently started to consider machine learning algorithms for side channel analysis, [102, 76].

[1]A code that is considered secure on one platform, may not be on another due to microarchitectural differences.

In this work we take another step in this direction. We are motivated by the need for automation in cross-VM leakage analysis. Our goal is to minimize the need for human involvement in formulating an attack. While more sophisticated techniques such as deep neural networks can solve more complicated problems, they require significantly more training data and take longer. Instead here we focus on more traditional ML techniques for classification. In particular, we are interested in automating classification of applications through their cache leakage profiles in the Cross-VM setting. A successful classification technique would not only compromise the privacy of a co-located user, but could also serve as the initial discovery phase for a more advanced follow-up high precision attack to extract sensitive information. To this end, in this work we first profile the cache fingerprints of representative benchmark applications, we then identify the minimal processing steps required to extract robust features. We train these features using support vector machines and report success rates across the studied benchmarks for experiments repeated for L1 and LLC. Finally, we take the attack to AWS EC2 to solve a specific problem, i.e. we use the classification technique to show that it is possible to detect other co-located VMs. We achieve this by sending ping requests to open ports by simultaneously monitoring LLC on Amazon EC2. If the ping receiver code is detected running on the co-located instance we infer co-location with the targeted IP.

Our Contribution

We present a study in automation of cache attacks in modern processors using machine learning algorithms. In order to extract fine grain information from cache access patterns, we apply frequency transformation on data to extracted fingerprints to obtain features. To classify a suite of representative applications we train a model using a support vector machine. This eliminates the need for manually identifying

patterns and crafting processing steps in the formulation of the cache attack. In our experimental work, we classify the applications bundled in the Phoronix Test Suite. Note that we do not have any information about the content of the code and nor have we studied any internal execution patterns.

In summary, this work

- for the first time implements machine learning algorithm, i.e. SVM, to profile the activity of other users on the cloud
- extracts the feature vectors from cache access data for different types of applications using a straightforward FFT computation,
- demonstrates that there is no need for synchronization between spy and target to profile an application while SVM based approach is implemented,
- shows that targeted co-location is achievable by sending ping requests on Amazon EC2 if the targeted IP is known by spy

The rest of the study is divided as follows. The approach is presented in Section 6.2. The experiment setup and results are explained in Section 6.3.

6.2 Methodology

In this section, we show how machine learning can be used on cache profiles to detect running programs. One specific use case is the detection of the ping service, which can serve as an implicit co-location test.

6.2.1 Extracting Feature Vectors from Applications on Cache

Our thesis is to show that programs have unique fingerprints in cache and it is possible to learn and classify application fingerprints using ML algorithms with a

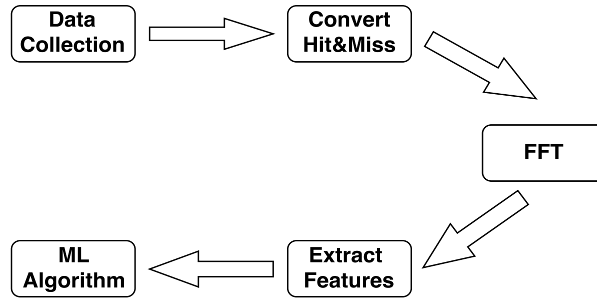


Figure 6.1: The flow chart of the approach for both L1 and LLC profiling

high accuracy. The proposed approach starts by creating profiles for every software using the Prime&Probe technique. This way, dynamic and static functions of the application are detected, resulting in fairly reliable fingerprints. The raw cache timing traces are first turned into hits and misses, followed by a Fourier transform. Performing a Fourier transform on the cache profiles removes the need for tight synchronization and makes the approach more resilient to noise. The FFT output can then directly be fed into a machine learning method of choice. The process to obtain application fingerprints is visualized in Figure 6.1. Our approach differs from previous works in cache-based information extraction in several ways:

Fourier Transform Most previous works [64, 81, 89], assume the monitoring process to be synchronized. The synchronization is handled by triggering the event, then the profiling phase is started. However, it is not trivial when the monitoring process and the target do not have communication. In addition, the functions periodically accessed in the application would give a certain information which could be exploited by using Fourier transform. Therefore, we transform the data to frequency domain from time domain in order to eliminate a strong assumption like synchronization and to extract the periodic functions' cache accesses as fingerprints of the applications.

No Deduplication Deduplication enables incredibly powerful side channel attacks [64,

29, 133], most prominently the Flush and Reload technique [172, 64]. However, public cloud service providers are aware of this issue and have disabled deduplication. Therefore, it is impossible to track data of other VMs in shared memory in IaaS and most PaaS clouds. Hence, the Prime&Probe technique is preferred to implement in our scenario instead of the Flush and Reload method to eliminate the strong assumption for deduplication. Prime&Probe technique is simply based on fill all ways in the monitored LLC set by enabling Huge pages which is possible in all public clouds.

The resolution of the resulting analysis is lower than Flush and Reload method in LLC, however the results show that after training enough data it is efficient to detect programs used by other co-located VMs.

Detecting Dynamic Code Our method does not make any assumption on whether code is dynamic, static or a shared function. Instead, we profile one of the *columns* in the cache, as shown in Figure 6.2. This means the location of a line in LLC might change from one run to another run if the function is dynamic. However, the offset bits (o) never change therefore, it resides on one of the set-slice pairs solving $s \bmod 64 = o$.

Long profiles Our method shows that even if the entire process of a program is not profiled, the spectral density of a small part of the program can give enough information to detect the program (in fact, the length of the analyzed programs varies from two seconds to 3.5 hours).

Our approach starts by creating cache profiles for every application by using the Prime&Probe technique in the same core to monitor all L1 cache sets. The analysis of L1 cache leakage provides a very high-resolution channel, thereby describing a best-case scenario for the learning technique. In addition, the L1 cache experiments

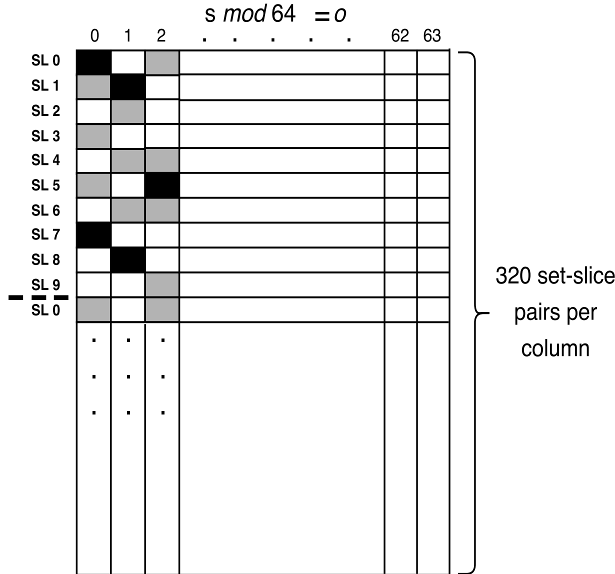


Figure 6.2: Visualization of 10 core LLC. Gray set-slice pairs are noisy, white set-slice pairs are unused sets and black set-slice pairs are actively used by target application.

provide valuable information which cache sets are actively used by the application. This information can be used as a preparatory step for LLC profiling to find the corresponding active sets in LLC. After the data is collected for a set of application, the Fourier transform is applied to extract the feature vectors subsequently used in an ML algorithm.

6.2.2 Extracting feature vectors from L1 cache

We assume that the number of L1-instruction and L1-data cache sets are S_{L1} for either cache. In L1 cache, the data and instruction cache are monitored separately, while profiling process and target application are running in the same core. The overall process to monitor L1 sets and creating feature vectors for different applications is given in Algorithm 2.

For L1-data and L1-instruction monitoring, a total of N_T traces is collected per set for each data set. Therefore, for each data set we have $S_{L1} \cdot N_T$ traces in total.

After collecting several data sets, the total number of traces is equal to $S_{L1} \cdot N_T \cdot N_D$ where N_D is the total number of data sets per application collected by the spy.

The outliers in the data should be filtered before the raw data (R) is converted to binary data (B). Hence, the L1-data and L1-instruction sets are monitored in idle case and base Probe values are recorded. The outlier threshold (τ_o) and binary conversion threshold (τ_c) are obtained based on the idle values.

Table 6.1: Symbol Descriptions

Symbols	Description
S_{L1}	Number of sets in L1 cache
N_T	Number of traces collected
N_D	Number of data sets per test
N_S	Number of applications
N_C	Number of cores
N_A	Number of active sets
τ_o	Outlier threshold for samples
τ_c	Hit&Miss threshold
F_s	Sampling Frequency
F_{CPU}	CPU frequency
T_{cc}	Prime&Probe time
L_f	Length of fingerprint

The Probe timings are compared to τ_o and τ_c. The values are higher than τ_o are set to median value of idle case of that set to get rid of the outliers. The conversion from R to B is also implemented by comparing with τ_c. If the Probe time is higher than τ_c, then the trace is converted to 1, implying an access to the cache set. If it is below than τ_c, the trace is converted to 0, implying no cache access. The resulting binary trace is converted by using Fourier transform.

In the transformation phase, the sampling frequency should be computed. In order to calculate the sampling frequency (F_s), the total Prime&Probe time for monitored set is computed in clock cycle (T_{cc}), then the CPU frequency (F_{cpu}) is divided by T_{cc} to get F_s for L1-data and L1-instruction cache. We assume that

the sampling frequency is same for all sets in L1-data and L1-instruction cache. To calculate the frequency components of binary data, F_s is used in FFT and the length of the outcome is N_T. However, the result has two symmetric sides of which only the first half is used as a fingerprint. Therefore, the length of a fingerprint obtained from one data set is $N_T/2$.

Algorithm 2 L1 Profiling Algorithm

$F_s = F_{CPU}/T_{cc}$
for i **from** 1 **to** N_S **do**
  for j **from** 1 **to** N_D **do**
    for k **from** 0 **to** $S_{L1} - 1$ **do**
      for l **from** 1 **to** N_T **do**
        if $R(i, j, k, l) \geq \tau_o$ **then**
          $R(i, j, k, l) = median(R(i, j, k, 1 : N_T))$
        end if
        if $R(i, j, k, l) \geq \tau_c$ **then**
          $B(i, j, k, l) = 1$
        else
          $B(i, j, k, l) = 0$
        end if
      end for
      $L(i, j, k) = FFT[B(i, j, k, 1 : N_T)]$
    end for
    $L_f^{i,j} = L(i, j, 0 : S_L - 1)$
  end for
end for

For each process there are S_{L1} different fingerprints hence, these fingerprints are concatenated from set 0 to set $S_{L1} - 1$ sequentially. Thus, the total length of the fingerprint of a process is $L_f = S_{L1} \cdot N_T/2$. If the total number of data sets is N_D then, the size of a training matrix of a process is $N_D \cdot L_f$.

The SVM then processes all matrices combined and labeled from 1 to N_S where N_S is the number of different applications. The final success rate is computed using 10-fold cross-validation.

6.2.3 Extracting feature vectors from LLC

Next, we apply the approach on LLC leakage. LLC has the advantage that is accessible for all processes on the same system. Hence, as long as the monitored process runs on the same system as the monitor, the side channel is accessible, even if the two processes run in different VMs.

After finding the most used L1 set, the corresponding sets in LLC should satisfy $s \bmod 64 = o$ where o is the L1 set number. The number of corresponding sets vary with the number of cores N_C. In total, the number of LLC set-slice pairs on current Intel CPUs can be determined by

$$S_{L3} = 2^{N_{LLCB}-N_o} \cdot N_C \tag{6.1}$$

where N_{LLCB} is the number of LLC bits and N_o is the number of offset bits. After the eviction set for each set-slice pair is created by using the algorithm [68], the Prime&Probe profiling starts. For LLC profiling N_T is same with L1 profiling and after N_T traces are monitored in one set-slice pair, the next set-slice pair is profiled. The reason behind this is to increase the temporal resolution for each set-slice pair which is crucial to catch dominant frequency components in frequency domain.

After collecting $N_T \cdot S_{L3}$ data, the same process in L1 profiling is applied to LLC traces to get rid of the outliers. Before the binary data is derived from the raw data, the noisy set-slice pairs need to be eliminated. For this purpose, the number of cache misses are calculated in idle case and if the number of cache misses are higher than 1% of N_T that set-slice pair is marked as noisy, as shown in Figure 6.3. After noisy sets are determined all of them are excluded from the next steps since the spectral density of these sets is not stable.

The active set-slice pairs are determined by checking the number of cache misses

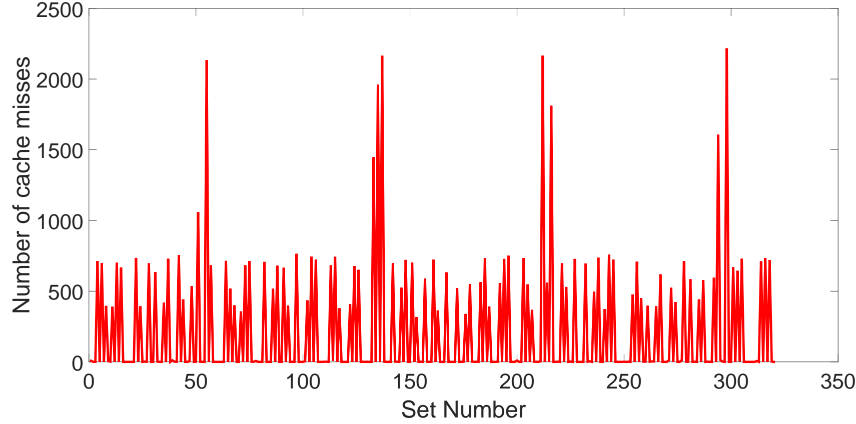


Figure 6.3: Eliminated noisy sets in LLC

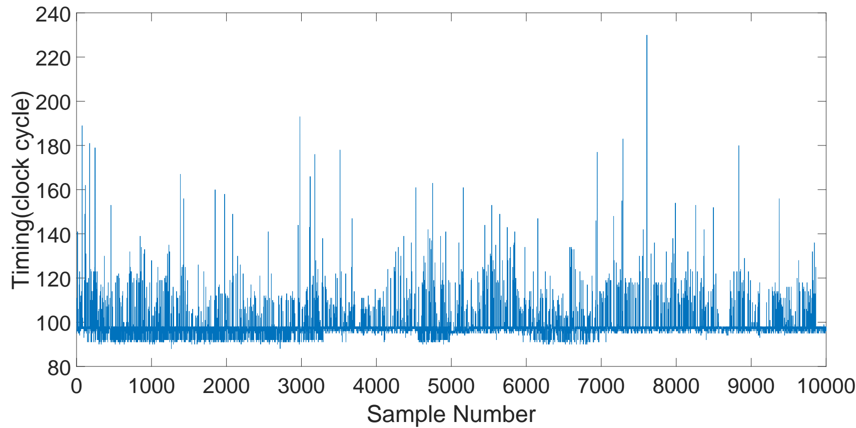


Figure 6.4: One of the active sets for an application

in the data. If the number of cache misses is higher than 3% of N_T in Figure 6.4, then the set-slice pair is marked as an active set. After all active sets are derived, they can be converted to binary data with the same process in L1 profiling in Figure 6.5 before the Fourier transform starts.

For the Fourier transform F_s should be calculated for LLC sets. F_s is lower in LLC profiling since the number of ways in the sets are higher than L1 sets and the access time to lines reside on LLC is greater than L1 lines. Therefore, the total Prime&Probe time for each set-slice pair should be calculated and the average of all of them are used as LLC F_s. After F_s is calculated, the active sets are transformed

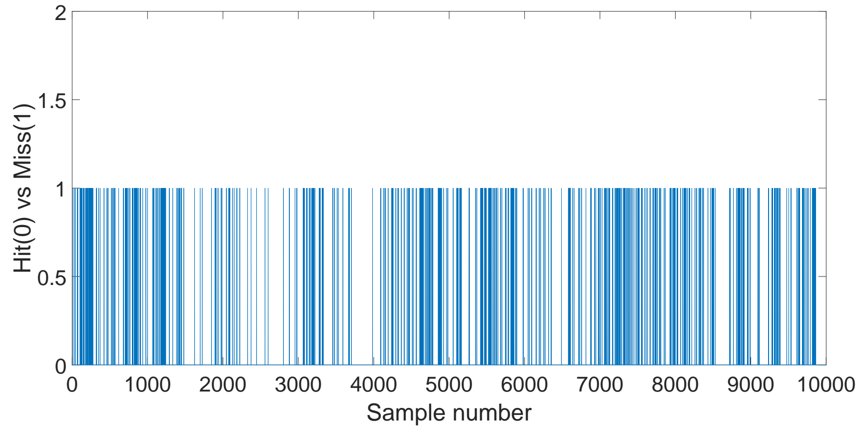


Figure 6.5: Hit(0) and miss(1) graph of an active set

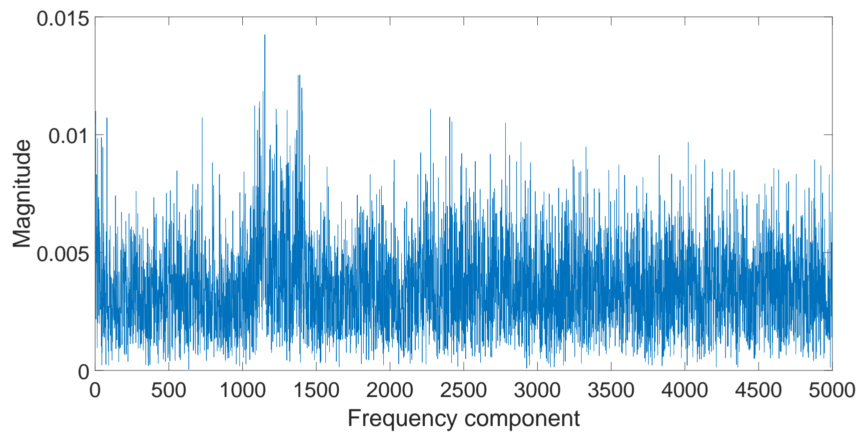


Figure 6.6: Frequency components of an active set

to frequency domain in Figure 6.6. The number of frequency components per N_T is same with the L1 profiling.

The number of active sets (N_A) may vary for each process therefore, the concatenated active sets have different length for each software. To solve this issue we propose to combine all frequency components of active sets. All frequency components are summed up element-wise and a fingerprint is obtained from each data set. In LLC profiling the length of the fingerprint is smaller than L1 profiling because in LLC scenario we cannot concatenate all active sets.

After obtaining all data sets for each application the total size of matrix for LLC

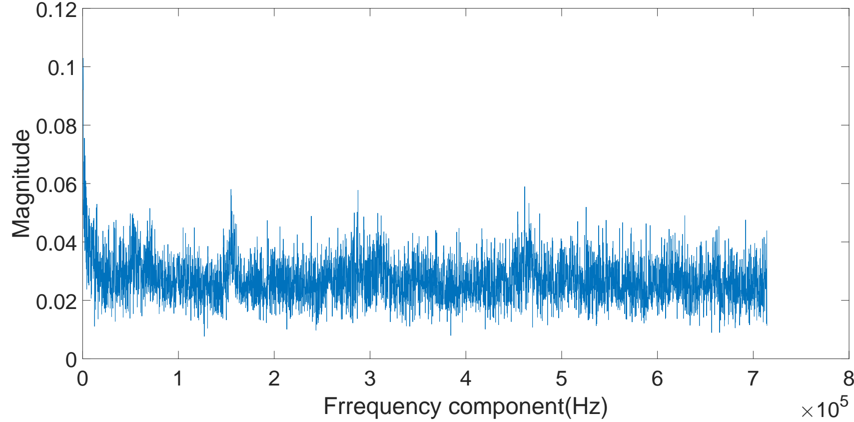


Figure 6.7: Combination of frequency components of all active sets

training data is $N_D \cdot (N_T/2)$. The SVM algorithm is applied as in the L1 profiling case and the results are recorded.

6.2.4 Targeted co-location by ping detection on the cloud

Another use case of the described methodology is the detection of whether or not a specific application is being executed. For this purpose we propose to detect ping requests sent to a target VM. We then try to detect the execution of the ping response process to verify and detect co-location with that target VM. In order to detect the co-location on the cloud, different types of covert channels such as LLC [82] and memory bus locking [158] have been used. These methods can be effective to verify the co-location between spy and target VMs. Our method also uses LLC, but, due to the omnipresence of ping support, this method is very widely applicable. The scenario is as follows: the spy VM monitors LLC sets by Prime&Probe to check the co-location with the target VM in the same cloud region. Another collaborating process of the spy sends ping requests to the target VMs with a certain frequency. These ping requests trigger executions of the ping service, which is then observable by the spy VM.

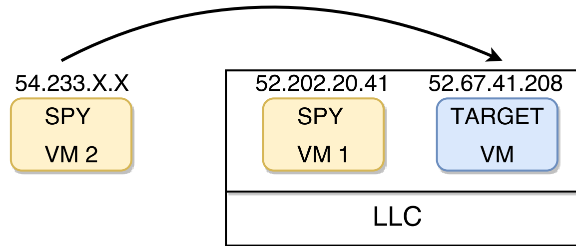


Figure 6.8: The scenario for ping detection on Amazon EC2

The used approach is similar to the previous cases: The monitored sets are determined by $s \bmod 64 = 0$. The reason behind this is the ping receptions are seen random sets. Therefore, we find that it is sufficient to monitor these sets to detect the ping. The steps to detect ping on the cloud are as follows:

1. Spy VM1 finds the noisy sets and excludes them from S_{L3} sets in VM1

2. Ping requests are sent by spy VM2 with a certain frequency

3. Spy VM1 begins to implement Prime&Probe on remaining sets

4. Spy VM1 determines the active sets in LLC

5. Fourier Transform is applied to the active sets

6. The frequency components are compared with the ping frequency

In our method, first the active IPs and the open ports should be found. In Amazon EC2 every region has different IP ranges. We focus on South America region and the IP range is documented [2]. Open ports can be found using the *nmap* tool.

There are two types of ping namely *hping* and *ping* commands. The *hping* command is more useful since specific ports can be pinged such as port 22 which is used for SSH connection. Furthermore, the frequency of ping requests for *hping*

command can be set higher than in the *ping* command. There is a need to have more frequent ping requests, as high-frequent calls to ping strengthen the LLC profile and thus decrease the number of traces needed to detect the ping requests in LLC. Therefore, we used *hping* in our Amazon EC2 experiments.

6.3 Application Detection Results

In this section, we explain the experiment setup used to collect data and make our scenario applicable.

6.3.1 Experiment Setup

For the experiments, we have used the following two setups;

- **Native Environment:** In this setup, the applications are running on a native Ubuntu 14.04 version with no virtualization. The processor is a 10 core Intel(R) Xeon(R) E5-2670 v2 CPU clocked at 2.50 GHz. The purpose of this scenario is to run experiments in a controlled environment with minimal noise and to show the high success rate of our methods. In addition, this processor is the same type of processor mainly used in Amazon EC2 cloud.

- **Cloud Environment:** In this setup, Amazon EC2 cloud servers are used to implement our experiments in a cross-VM scenario. In Sao Paulo region, the processors are same with the one used in native environment with a modified Xen hypervisor. The instance type is *medium.m3* which has 1 vCPU. The aim of this setup is to show the huge thread of our scenario in a public cloud. In this setup, there are two co-located VMs in the same physical machine sharing the LLC which is verified by the techniques [82].

To evaluate our approach on a broad range of commonly used yet different applications, we decided to use the Phoronix test benchmarks as sample applications for classification [10]. We performed classification experiments on these applications in three different scenarios. As baseline experiments we first performed the experiments in the above-described native scenario, both by monitoring L1 cache leakages and also by monitoring LLC leakages. The former shows the potential of L1 cache leakages if they are accessible. The latter assumes a realistic observation scenario for any process running on the same system. Finally, we performed the same experiments on Amazon EC2 cloud to show the feasibility in a noisy and cross-VM scenario. In this public cloud scenario, only LLC is profiled to classify benchmarks since each VM has only one thread in the core and they do not reside on the same core. For both L1 cache and LLC experiments, our methodology is applied to 40 different Phoronix benchmark tests in including cryptography, gaming, compressing, SQL, apache and so on Appendix. Last but not least, we present a scenario where we only try to detect the presence of a single application, the ping detection described in Section 6.2.4.

6.3.2 Application Detection in Native Environment

We first performed experiments in the native environment.

6.3.2.1 Monitoring L1 Cache

In native case, first we implemented our profiling on L1 cache. There are two types of cache structure namely, data and instruction. Therefore, in our experiments we profiled each of them separately. In our processor there are $S_{L1} = 64$ sets for each L1-data and L1-instruction cache and the sets are 8 way associative.

The profiling and application code run on the same core to detect misses in

L1 cache. Hence, the hyper-threading feature of Intel processors is used. Before the training data is collected, an idle case of L1-data and L1-instruction sets are monitored and base Probe values are recorded. For L1-data the base value is around 65 clock cycles and for L1-instruction it is around 75 clock cycles. Hence, the outlier threshold is chosen as $\tau_o = 150$ for both data and instruction cache. For the conversion from raw data to binary data the threshold value is $\tau_{o,d} = 80$ for data cache and $\tau_{o,d} = 90$ for instruction cache. The number of traces collected per set for each data set is $N_T = 10,000$. Therefore, the total number of traces is equal to 640,000 which belongs to one data set for L1-instruction or L1-data.

To compute the sampling frequency, we checked the total Prime&Probe time and it is almost same for all sets in L1 cache which is around $T_{cc} = 200$ clock cycle. Hence, the sampling frequency is $F_s = 2.5GHz/200 = 12.5MHz$ for L1 cache profiling. F_s for L1 cache is higher than LLC profiling because the number of ways in L1 sets is smaller than LLC sets and accessing to L1 cache lines is faster than LLC lines. Thus, the resolution of L1 profiling is higher than LLC profiling which results more distinct feature vectors and high success rates in ML algorithm.

After F_s is determined, FFT can be applied to traces. The outcome of FFT is $N_T/2$ which is equal to 5,000 frequency components in our case. This process is applied to all 64 sets in data and instruction cache for each test. Hence, the feature vector of a test consists of 320,000 frequency components after all sets are concatenated. The number of data sets per test is $N_D = 60$ which means the training data is a matrix of the size $2,400 \times 320,000$.

To classify the training data, first 10-fold cross validation is implemented in SVM. For cross-validation we implement both C-SVC and nu-SVC SVM types in the LIBSVM library. Our results show that C-SVC gives better success rates, so we preferred this option. For the kernel type, the linear option is chosen since the

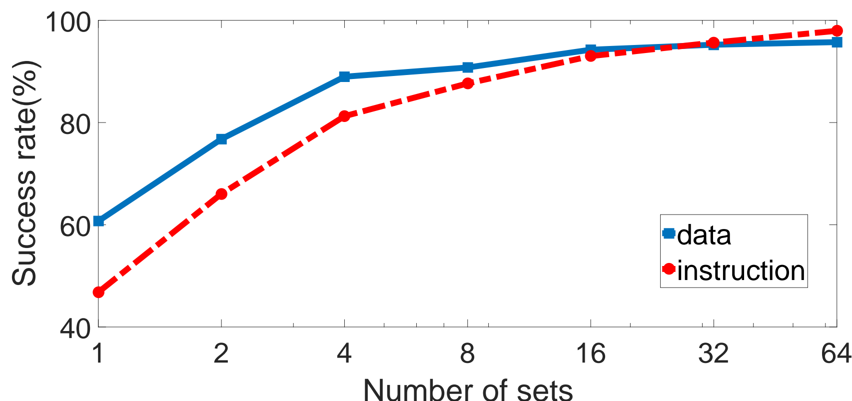


Figure 6.9: Success rate graph for varying number of sets to train the data

success rate is much higher than for the other options. After these options are chosen kernel parameters and the penalty parameter for error are optimized by LIBSVM. In both training and test phases the chosen parameters are used to implement SVM. Therefore, there is no user interaction to choose the best parameters and the steps are automated.

In cross-validation experiments, we show the effect of number of L1 sets on success rate. If only 1 set is used to generate the training model, the cross-validation success rate is 46.8% for instruction and 60.71% for data cache. With the increasing number of sets, the cross-validation success rate for data and instruction cache is increasing to 95.74% and 97.95%, respectively in Figure 6.9.

For training and individual success rate of test, 60 data sets per test are trained where the SVMMODEL is obtained with C-SVC and linear kernel options. With the cross-validation technique, the success rate for instruction cache is higher than data cache. The reason behind this is some of the Phoronix tests do not use L1-data cache however all tests use L1-instruction cache. Therefore, extracting the feature vectors for tests in instruction cache is more successful than L1-data cache.

The results also show that the cross-validation success rate is 98.65% if all information in L1 cache (both instruction and data) is used in the machine learning

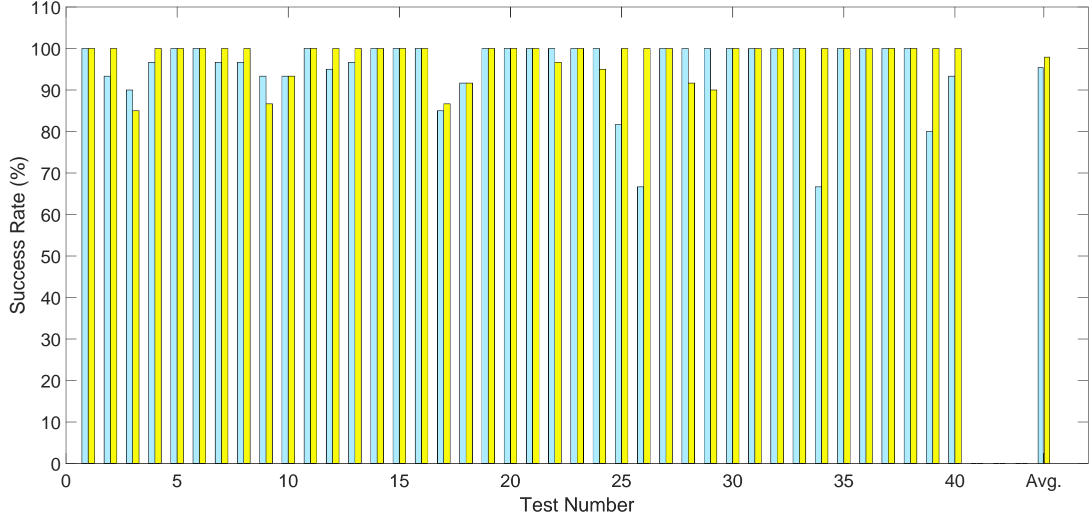


Figure 6.10: Success rate for different tests in L1-data (blue) and L1-instruction (yellow). The last bar represents the average of success rates for 40 tests

algorithm. To achieve this success rate we used all 64 cache sets and in total we have $50 \times 640,000$ size feature vectors per test. Therefore, the size of training data is $2,000 \times 640,000$.

6.3.2.2 LLC Results

L1 profiling is not realistic in the real world since the probability of two co-located VMs in the same core is really low. Therefore, before switching to public cloud we implemented our attack in LLC with a cross-core scenario. The number of cores is $N_C = 10$ in our processor and the number of set-slice pairs solving the equation in 6.1 $s \bmod 64 \ = \ o$ is $S_{L3} = 2^5 \cdot 10 = 320$ where $N_{LLCB} = 11$ because of 2,048 LLC sets in total and the number of offset bits is $N_o = 6$. o is the set number which is the most used one in L1 profiling for that test. Therefore, we have 320 set-slice pairs in total to monitor.

Before collecting data for every test, the idle case of each set is monitored to determine the base value (τ_b). τ_b changes between 90 and 110 clock cycle among

different set-slice pairs. Hence, for each set-slice pair τ_b is different. The outlier threshold (τ_o) is 250 clock cycle. The threshold value (τ_c) for the conversion from raw data to binary data is $\tau_b + 15$ clock cycle. After obtaining the binary data, it is trivial to find the noisy sets. If the number of cache misses is higher than 100 in a set-slice pair, it is marked as noisy. These noisy sets are not processed when the data is collected.

While collecting the training data 10,000 traces are collected per set-slice pair. The active sets are determined by checking the number of cache misses in each set-slice pair excluding the noisy sets. If the number of cache misses is higher than 300, then that set-slice pair is marked as active and they are included in Fourier transform.

The Prime&Probe timings change between 1,800 and 2,200 clock cycle so the sampling frequency (F_s) is taken 1.3 MHz. After FFT is applied to active sets, the left symmetric side of the outcome is recorded. The length of the fingerprint for a set-slice pair consists of 5,000 frequency components. If there are 6 active sets for a test, the fingerprint of each active set are combined by element-wise in each data set and final fingerprint is obtained from one data set per test.

For LLC experiments, we used 40 different benchmark tests to profile in LLC. The number of data set per test is 50 and the length of vector for each feature vector is 5,000. After collecting the data the training model is generated and the cross-validation is applied to training data.

For the cross-validation, same options in L1 profiling are used in SVM. The success rate for LLC test in average is 77.65% with 5,000 frequency components. With the decreasing number of frequency components the success rate drops to 45% in Figure 6.11.

The details of success rates for different tests are presented in Figure 6.12 by

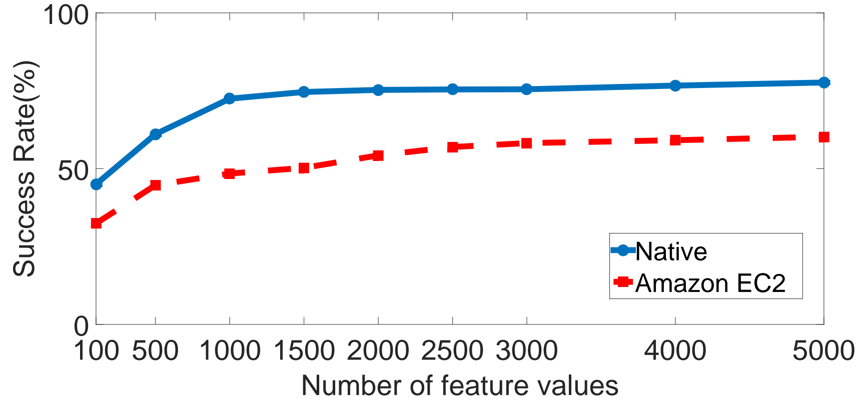


Figure 6.11: LLC success rate with varying number of frequency components

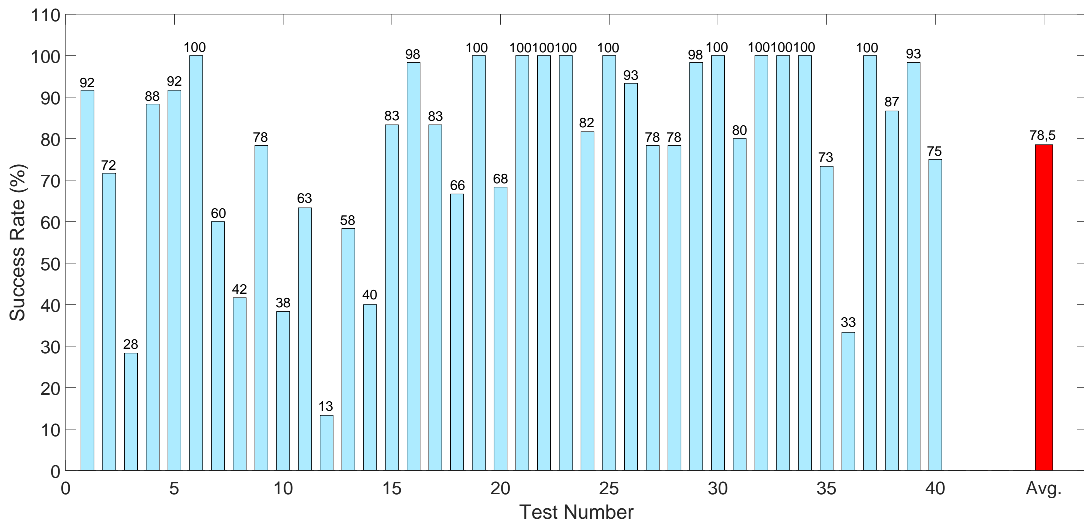


Figure 6.12: LLC success rates for different tests in native scenario. The blue bar represents the success rate for different tests. The last bar shows the average success rate for all tests

using 10-fold cross-validation technique. The results are obtained from 5,000 frequency components and 60 data sets per test. The lowest recognition rate is 13% for *GMPBENCH* test since the success rate for this test is low in L1-data cache in Figure 6.10.

6.3.3 Application Detection on EC2 Cloud

To show the applicability of ML technique to real world, we also perform our profiling method on Amazon EC2. The challenges of performing the experiments on a public cloud are hypervisor noise and the noise of other VMs in the monitored sets. Therefore, some set-slice pairs are marked as active even if those pairs are not used by target VM. Redundant cache misses in the active sets also pose a problem. During Fourier Transform, these cache misses may cause shifts in frequency domain. To overcome these difficulties, SVM technique is applied to the data, and as a result, the success rate gets higher.

The number of tests decreases in cross-VM scenario since some tests do not work properly and some of them have installation problems on Amazon EC2. Thus, the number of tests used in this experiment decreases to 25. To classify the different benchmark tests, same process in LLC profiling is used, then training data is processed in SVM. The result is lower than native case because of the aforementioned types of noise. The 10-fold cross-validation result is 60.22% in Figure 6.11 with 5,000 frequency components. This result shows that on public cloud the classification success rate drops with increasing noise.

The success rates for individual tests change between 16% and 100% in Figure 6.13. The success rate decreases when the hypervisor and other VMs noise affect the cache miss patterns. Even though the success rate is lower than native scenario, this result demonstrates the applicability of out method in the cloud platform.

6.3.4 Ping detection on EC2

To detect the co-located VMs with spy VM, ping requests are sent by one of the VMs controlled by the spy in the same region. The purpose of this is to decrease

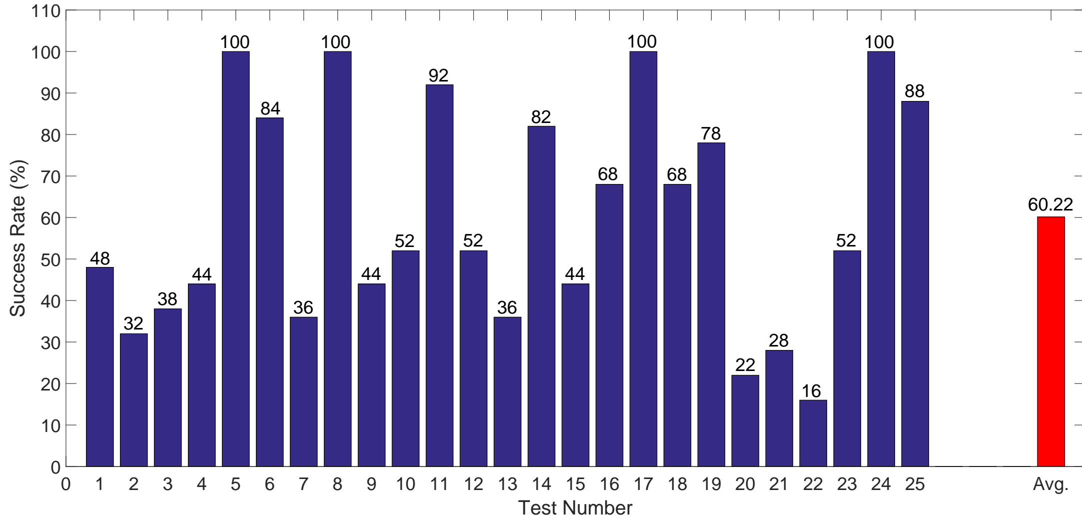


Figure 6.13: LLC success rates for different tests in cloud scenario. The blue bar represents the success rate for different tests. The last bar shows the average success rate for all 25 tests

RTT and increase the frequency of ping requests. At the same time, spy VM 2 monitors 320 set-slice pairs since the processor has 10 slices and 32 different set numbers satisfying $s \ mod \ 64 \ = \ 0$.

The set-slice pairs are very noisy on the cloud therefore even if the candidate VM is not co-located with the spy VM, there are some active sets in LLC because of the noise from other VMs. However, when the frequency domain of active sets is checked by the spy, there is no dominant frequency component or the dominant frequency components are not consistent with the ping frequency. If the target VM is co-located with the spy VM, then the periodic cache misses can be seen in one of the active sets in Figure 6.14.

After applying Fourier Transform with an appropriate F_s, the dominant frequencies are clearly seen in Figure 6.15. In order to calculate the frequency domain the sampling frequency F_s should be computed before the frequency transformation is applied to the data. After averaging all LLC sets, F_s is determined to be around 1,800 clock cycle. The normal CPU frequency of the processor is 2.5 GHz so F_s is

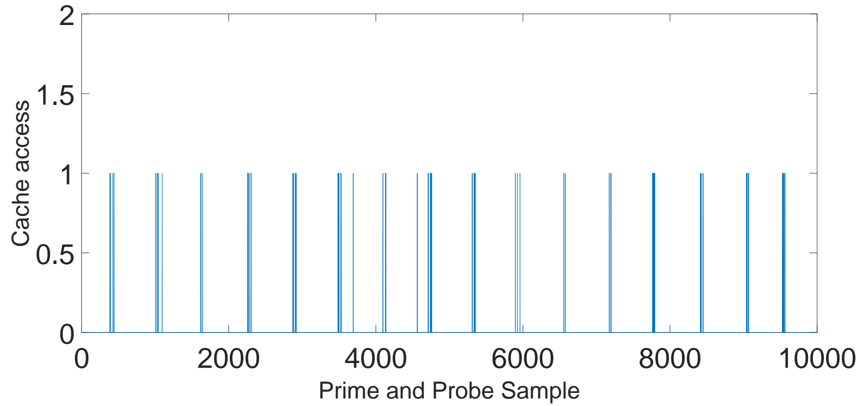


Figure 6.14: Cache miss pattern of received ping requests in LLC

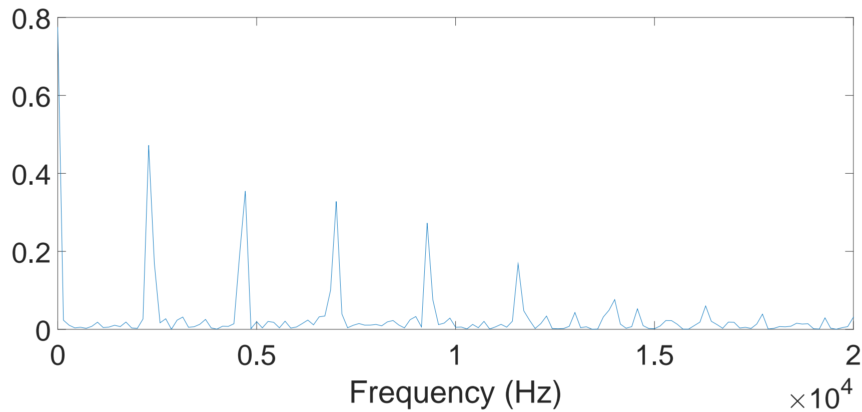


Figure 6.15: Frequency components of ping requests in LLC

equal to 1.56 MHz on Amazon EC2 VMs. When the ping requests are sent every 0.4 ms from Spy VM2, then Spy VM1 can monitor the cache misses in active sets as in Figure 6.14. When the frequency domain is generated, the frequency components overlap with the frequency of ping requests in Figure 6.15.

6.4 Conclusion

In this paper we tackled the problem of automating cache attacks using machine learning. Specifically, we devised a technique to extract features from cache access profiles which subsequently are used to train a model using support vector machines.

The model is used later for classification applications based on their cache access profiles. This allows, for instance, a cloud instance to spy on applications co-located on the same server. Even further, our technique can be used as a discovery phase of a vulnerable application, to be succeeded by a more sophisticated fine grain attack. We validated our models on test executions of 40 applications bundled in the Phoronix benchmark suite. Using L1 and LLC cache access profiles our trained model achieves classification rates of 98% and 78%, respectively. Even further, our model achieves a 60% (for a suite of 25 applications) in the noisy cross-VM setting on Amazon EC2.

Chapter 7

Machine Learning based Website Detection

7.1 Motivation

Web browsers have become indispensable components in our lives: They provide access to news and entertainment. More importantly, browsers have become the de facto platform through which we perform privacy and security sensitive interactions such as online banking, web enabled healthcare, social networking etc. Due to numerous vulnerabilities sandboxing has become a standard feature across browsers, offering isolation from other sites viewed and other processes executing in the OS. To thwart remote tracking, browser also commonly implement an *incognito* mode. In the extreme case we have Onion routing, i.e. Tor, enabled browsers which allow users to protect their privacy against Internet surveillance by randomizing the routing algorithm to hide the path packets take. By using a Tor browser a user may hide the websites she visited. Such tools have become indispensable for whistleblowers and dissidents who try to protect their identity against powerful corporations

and repressive governments. Besides privacy preserving browsers, other tools have emerged to mask the identity of the user, e.g. Signal/Redphone, Silent Phone and Telegram. Even the installation of such tools can be viewed as subversive action by a repressive regime. In contrast, privacy enabled browsers come pre-installed on most platforms. While browsers have significantly matured in providing privacy assurances, they are still far from perfect. For instance, an attacker can still compromise the users privacy by exploiting microarchitectural leakages at the hardware level. In 2012, Jana et al. [94] found out that memory footprints of processes are unique and they could be used to detect the visited websites in the victim machine. In 2015, Liu et al. [111] profiled the whole LLC in the system and Oren et al. [127] implemented this technique to detect a small set of webpages in Javascript. What enables such attacks is that many of the applications we use every day run in the background in user space. Users trust these applications, even though they have little control over what is executed by third-parties.

In this work, we show that it is possible for such a third party application to collect data using hardware performance events (HPEs) and infer private user activity across application boundaries, e.g. to track visited websites, even if privacy-protecting technology, such as Incognito mode or the Tor browser are used. Such malicious behavior is straightforward in Linux, as the `perf` subsystem of the kernel allows to monitor HPEs from user space. Since the used HPE side channel information is incidental and often noisy, advanced methods for data analysis are needed. The recent advances in Machine Learning (ML) provide us with a powerful tool to classify the complex noisy data in an effective manner. We show that while SVM, Decision Tree and other classic ML methods are not sufficient to classify the complex and noisy observed data into a high number of different classes, methods such as Convolutional Neural Networks (CNN), a Deep Learning technique, can efficiently

extract meaningful data even in the presence of severe noise. By comparing the various ML methods on data collected by a malicious user space process monitoring HPEs with `perf`, we show that with CNNs, private user information can be inferred with very high success rates in a highly automated fashion even if precautions such as Tor browser are used.

Our Contribution. In summary, in this work:

- We combine advanced Machine Learning techniques to classify websites and compare the efficiency of ML techniques in various scenarios.
- We use `perf` to access different types of hardware based side channels in the system and combine them to get a better classification rate.
- We cover a wide-range of websites to classify (40 different web-pages) in the victim machine, including 30 of the top ALEXA sites and 10 whistleblowing web sites. In addition, we detect different subdomains in a domain to show that finer-grained and leveled browsing tracking is possible.
- We demonstrate that the attacker does not need to synchronize with the browser, as the alignment process is handled by the ML techniques.
- It suffices to monitor events for 1 second in Google Chrome and 5 seconds in Tor browser scenarios to classify the websites with high accuracy.

7.2 Browser Profiling Scenarios

We investigate the inference of opened websites via HPEs in three distinct scenarios hosted on two Linux test systems. The first system features an ARM Cortex-A53 processor with six programmable hardware counters, the second one comprises an

Intel i5-2430M processor with three programmable counters. Given the limited number of counters on both systems, only a selection of HPEs is measured in each experiment. A complete list of events supported by `perf` can be obtained from the Linux man-pages[106]. As we are relying on the standardized `perf_event_open` system call of the Linux kernel, there is no need to change the measurement code when switching between systems. Further details about the profiling scenarios are given in the following paragraphs.

Google Chrome on ARM. In this scenario, we profile the Google Chrome browser (v55.0.2883) with default options on the ARM system. While the browser loads websites, a malicious user space application is measuring six hardware performance events: `HW_INSTRUCTIONS`, `HW_BRANCH_INSTRUCTIONS`, `HW_CACHE_REFERENCES`, `L1_DCACHE_LOADS`, `L1_ICACHE_LOADS`, and `HW_BUS_CYCLES`. This selection of events covers instruction retirements, cache accesses, and external memory interfaces. It gives a comprehensive view of the microarchitectural load the browser is putting on the processor. The selected events are measured core-wide, hence including noise from other processes and background activity of the operating system. Since we want to assess the feasibility of core-wide profiling, the browser process is bound to the measured processor core. The events are then measured for five seconds.

Google Chrome (Incognito) on Intel. In this scenario, we profile Google Chrome in Incognito mode with default options on the Intel system. Since the number of counters is limited to three on this processor, the malicious user space application is measuring only three events: `HW_BRANCH_INSTRUCTIONS`, `HW_CACHE_REFERENCES`, and `LLC_LOADS`. Moreover, the events are acquired in a process-specific fashion, hence the browser processes float on all processor cores. The events are then measured for one second specifically for the rendering process of the opened website. Compared to the ARM scenario, the reduced event selection still provides

a meaningful view of the microarchitectural load. As the browser processes are not bound to one core anymore, we substitute events related to the L1 cache with last-level cache loads. In addition, the bus cycle event is omitted, because it is noisier on the Intel platform. Also, overall retired instructions are omitted, because we found the retired branch instructions to yield more usable information.

Tor Browser on Intel. In this scenario, we profile the Tor Browser (v6.5.1, based on Firefox v45.8.0) on the same Intel platform as before. In contrast to Chrome, the Tor Browser renders all tabs in one process, which is subsequently profiled by the malicious application. While the same three performance events are observed, the measurement duration is prolonged to 5 seconds. This is because the Tor network introduces significant delays while opening websites.

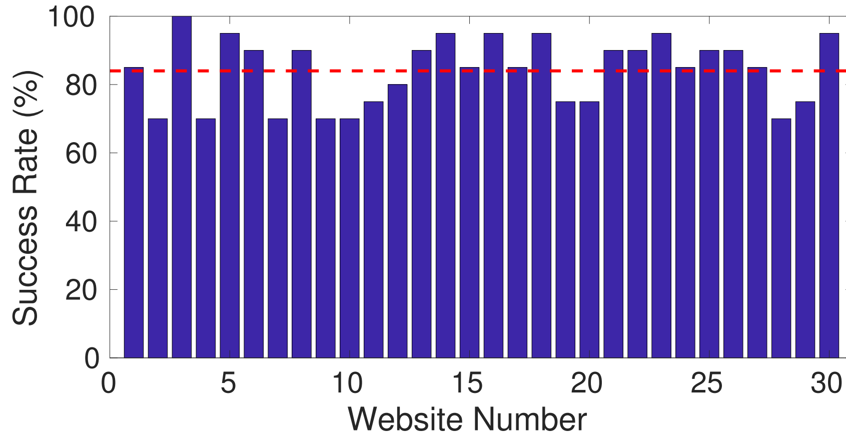
Synchronization. None of the scenarios require strict synchronization between the browser and the malicious application. Small misalignment is simply passed on to the Machine Learning step. Therefore, we only investigate simple synchronization techniques that can be achieved in practice. For Google Chrome on Intel, the adversary scans the running processes twice per second and checks whether a new rendering process has been spawned. Once a new process is detected, the adversary starts to measure the corresponding process-specific events. The Tor Browser, in contrast, is started freshly for every opened website. Again, the adversary checks all running processes twice per second and once the Tor Browser is detected, the process-specific profiling is started. This includes additional noise as the browser startup phase is also captured. In the ARM scenario, the measurements are precisely aligned with the start of loading a website. This is used to investigate whether more precise alignment yields better results. Such a trigger signal could be derived from a sudden change or characteristic pattern in the event counts, as the load of the system changes when a website is opened.

7.3 Website Profiling Results

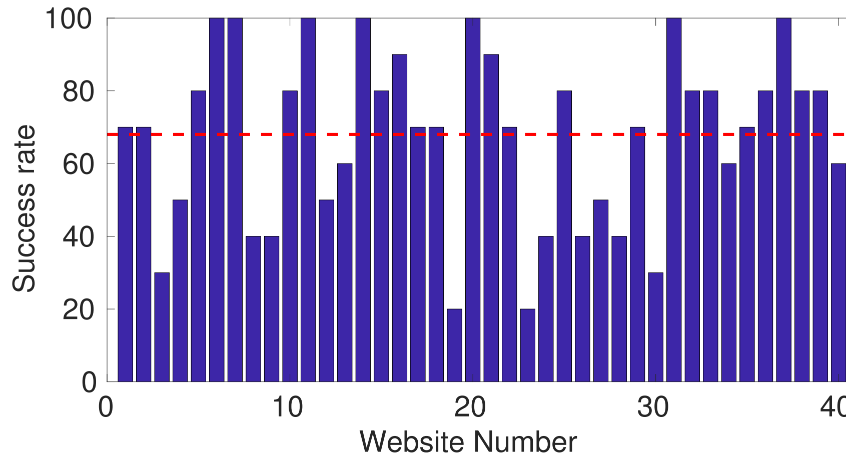
In each of the profiling scenarios described in Section 7.2, we monitor events when loading the homepages of the most visited websites according to Alexa [19]. The tested websites, excluding adult ones, is listed in Appendix A.1 (1-30). This illustrates the general effectiveness of the Machine Learning techniques to classify websites based on HPEs. To demonstrate that also fine-grained profiling is feasible, 10 different sub-pages of the `Amazon.com` domain are monitored in Google Chrome on Intel. Finally, a selection of whistleblowing websites is measured when visited with the Tor browser. They are also given in Appendix A.1 (31-40).

Google Chrome on ARM. For the experiments on ARM, each website is monitored 20 times to train the models. For each of these visits, 25,000 samples are acquired per hardware performance event. The samples of all events are then concatenated to yield a final measurement size of 150,000 samples. For 30 websites, the total training data size is therefore $90 \cdot 10^6$ samples. Based on this training set, the success rates after cross-validation are 84% for linear SVM, 80% for kNN, and less than 50% for DT and CNN. The low success rates of DT and CNN indicate that not enough samples have been acquired. Figure 7.1(a) illustrates the success rates for each of the visited websites when classified with SVM. Since the number of samples collected in this scenario is small, 10-fold cross-validation is used. The lowest detection rate is 70%, which shows that core-wide profiling is feasible even in the presence of background noise. The average classification rate of 84% is shown as a dashed line in the figure.

Google Chrome (Incognito) on Intel. For the Google Chrome experiments on Intel, the number of measurements per website is increased to 50. As more samples are acquired, fixed training and test sets are derived instead of using cross-



(a) SVM Success Rates

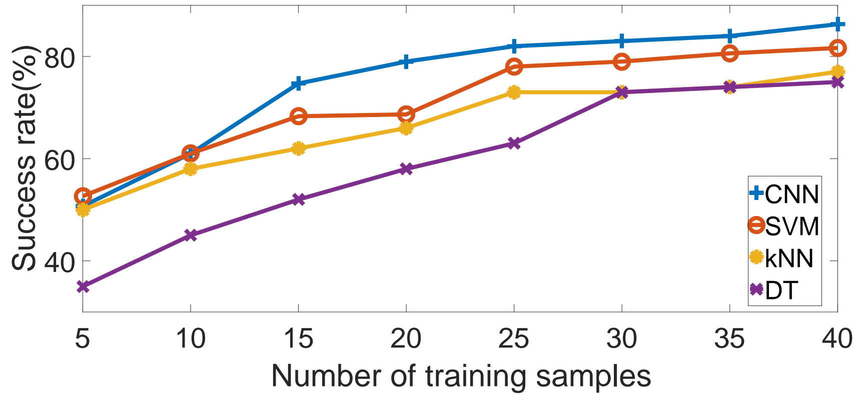


(b) CNN Success Rates

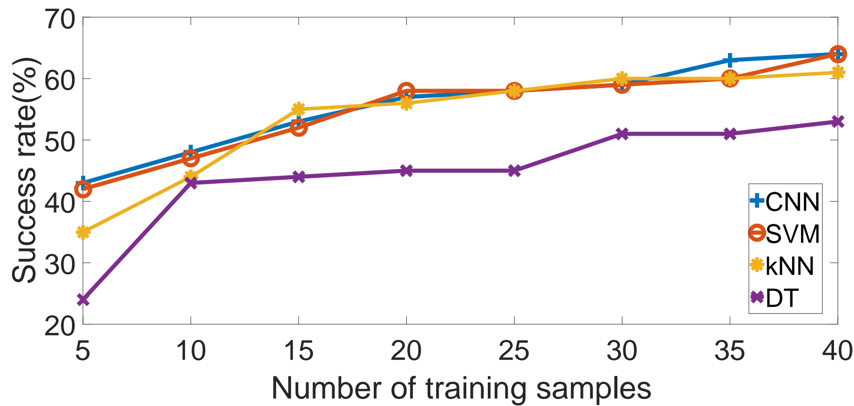
Figure 7.1: Success rates per website for (a) Google Chrome on ARM with the dashed line showing an average classification rate of 84%, and (b) Tor Browser on Intel with the dashed line showing an average classification rate of 68%.

validation. Out of the 50 observations, 40 are used for the training phase whereas 10 are collected to test the derived models. Since each website is monitored for only one second, every measurement now consists of 10,000 samples per event. With three observed events, this yields a total training set size of $36 \cdot 10^6$ and a test set size of $9 \cdot 10^6$ samples.

Figure 7.2(a) shows the success rates over an increasing number of training measurements for all Machine Learning techniques. Clearly, CNN achieves the highest



(a) Alexa Top 30

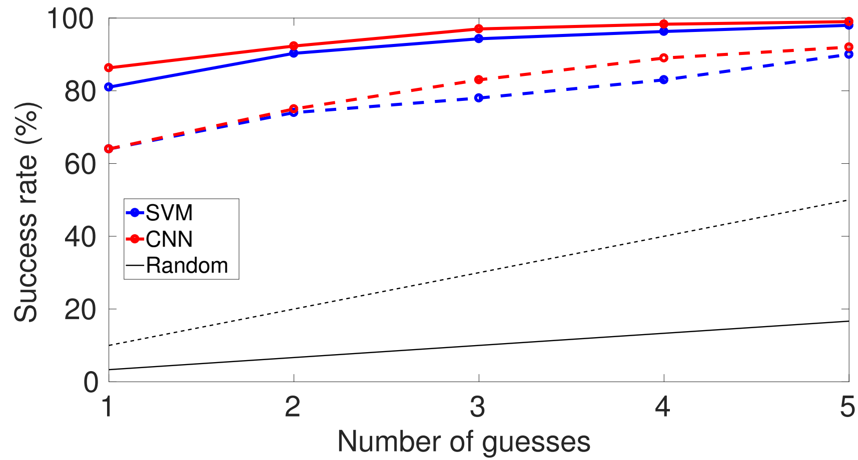


(b) Same Domain Pages

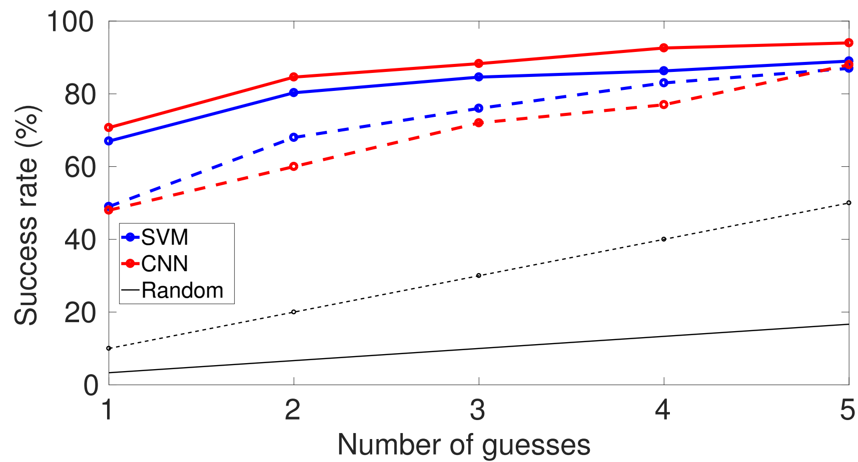
Figure 7.2: Success rate vs. number of training measurements for Google Chrome (Incognito), and (a) 30 different websites (b) 10 same domain web pages.

classification rate, if enough training samples are available. In particular, the success rate for 40 training observations per website is 86.3%. If the training data size is small, SVM and kNN achieve similar success rates as CNN. Due to the large size of feature vectors in the training and test data, DT gives lower success rates than other ML techniques. Regarding the computational effort, the training phase of CNN takes 2 hours on a GPU and is consequently the longest among the Machine Learning techniques. In contrast, the test phase takes approximately 1 minute for every ML technique.

The second experiment for Google Chrome in Incognito mode on Intel assumes



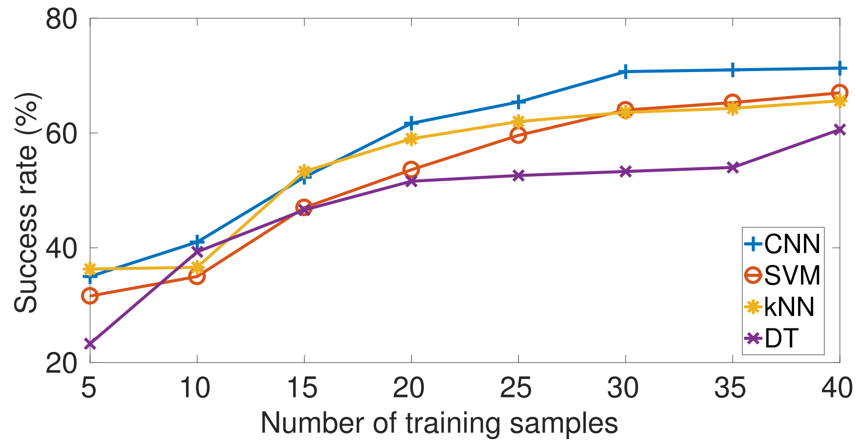
(a) Google Chrome (Incognito)



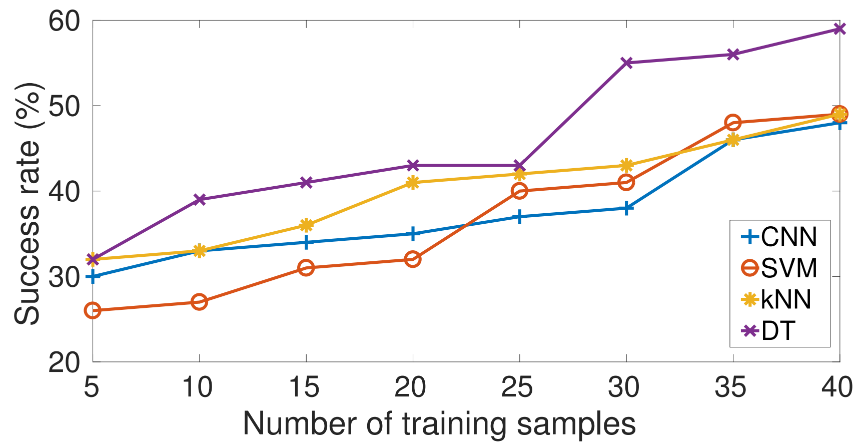
(b) Tor Browser

Figure 7.3: Number of guesses vs. classification rate for (a) Google Chrome (Incognito), and (b) Tor Browser. Solid lines represent results for Alexa Top 30, while the dashed lines illustrate the same domain results.

that an adversary has detected a website that the user has visited. Consequently, the attacker tries to infer which page of the website the user is interested in. To illustrate the feasibility of this attack, we selected 10 pages of the `Amazon.com` domain that display different sections of the online store (kitchen, bedroom, etc.). Naturally, this scenario is more challenging, as the difference between web pages of the same domain is smaller than for entirely different websites. Nevertheless, it is still possible



(a) Alexa Top 30

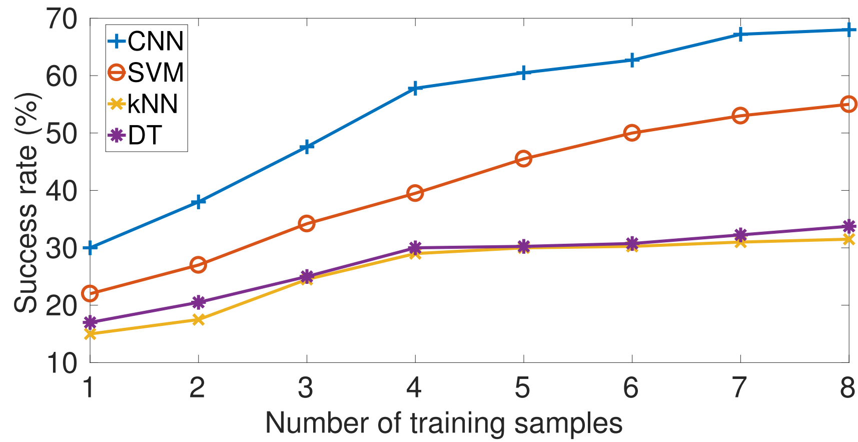


(b) Same Domain Pages

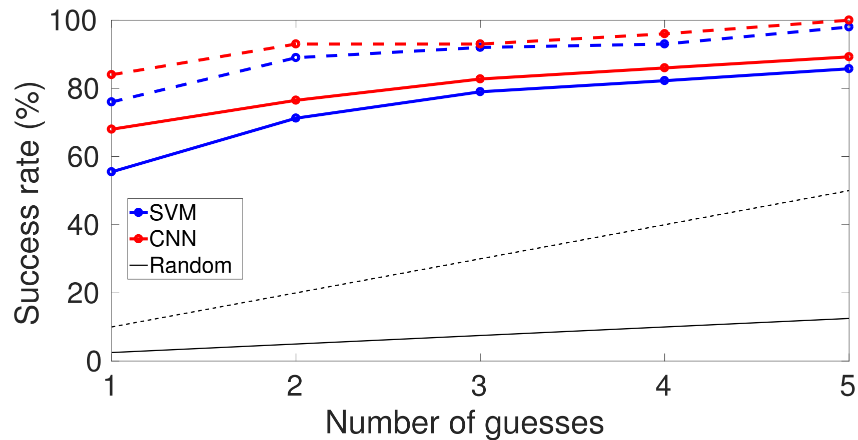
Figure 7.4: Success rate vs. number of training measurements for the Tor Browser and (a) 30 different websites, or (b) 10 same domain web pages.

to correctly classify the visited web pages with moderate success. This is illustrated in Figure 7.2(b). When using CNN and SVM, the success rate is 64%. kNN yields 60% success rate, while DT drops to 52%. For CNN and SVM, we also investigate the success rates when the number of guesses is increased in Figure 7.3(a). If the first 5 result classes are considered, websites can be detected with 99% accuracy for SVM and CNN. Similar results are obtained for the same domain experiments, where both CNN and SVM yield 92% accuracy.

Tor Browser on Intel. For the Tor Browser experiments, the same events as



(a) All Websites



(b) Whistleblowing vs. Others

Figure 7.5: (a) Success rate vs. number of training measurements for Tor Browser and all websites. (b) Number of guesses vs. classification rate for whistleblowing (dashed) and all websites (solid).

before are observed as well as the same number of measurements are taken for each website. Again, 40 of those measurements are used to construct the training set, while 10 measurements form the test set. As the Tor Browser is monitored for 5 seconds, 50,000 samples are acquired for each event and website. This yields 150,000 samples for one measurement, $180 \cdot 10^6$ samples for the entire training set, and $45 \cdot 10^6$ samples for the test set.

Figure 7.4(a) shows the success rates over an increasing number of training mea-

surements for all Machine Learning techniques. CNN yields the highest success rate of 71%. While SVM and kNN have similar success rates around 66%, Decision Tree yields a lower accuracy of 60%. The results show that CNN can handle noisy data and misalignment problems better than other methods, since CNN learns the relations between traces. The experiment results for the 10 web pages on `Amazon.com` are illustrated in Figure 7.4(b). In contrast to the Google Chrome results, Decision Tree yields the highest success rate of 59%. We believe the reason is the small number of classes that increases the efficiency of DT. The remaining algorithms classify the same domain web pages with a similar success rate of approximately 49%. In addition, Figure 7.3(b) shows the success rates for CNN and SVM over an increasing number of guesses. While the random selection success rate is around 16% for 5 guesses, CNN achieves a success rate of 94%. For the same domain web pages, the success rate of CNN is 88% for 5 guesses. SVM achieves slightly worse results.

Finally, we investigate whistleblowing websites, as visiting them anonymously is an important reason to use the Tor Browser. For this experiment, we select 10 whistleblowing portals from [7] (also given in Appendix A.1). In the first step, these websites are classified using all ML techniques. While CNN yields the best classification rate of 84%, SVM exhibits a success rate of 78%. In contrast, DT and kNN have lower success rates around 60%. In the second step, the classification is repeated for all websites considered so far (whistleblowing and Alexa Top 30). Figure 7.5(a) illustrates the success rates for all ML techniques. When classifying 40 websites, CNN yields a success rate of 68%, while SVM achieves 55%. In contrast, kNN and DT algorithms cannot classify the websites effectively. When the number of guesses is increased, the success rate improves again. Figure 7.5(b) shows the classification rates over an increasing number of guesses. If only whistleblowing websites and 5 guesses are considered, CNN yields a success close to 100%. When

all websites are considered, the success rate of CNN is 89.25%. SVM achieves slightly worse results. Individual success rates for CNN are shown in Figure 7.1(b). The lowest success rate is around 20% for two websites and seven websites are classified correctly with 100% accuracy. An interesting observation is that among the 40 websites, the whistleblowing portals are still classified with good success rates. With an average success rate of 68%, CNN is more capable than other ML techniques to correctly classify websites opened in Tor browser.

7.4 Discussion

The experiments on ARM were conducted with core-wide measurements, whereas HPEs were acquired in a process-specific fashion on Intel. In general, core-wide acquisition is expected to introduce more noise in the measurements, e.g., from system activity in the background. For process-specific acquisition the activity of the rest of the system does not impair the measurements, as the `perf` subsystem accumulates event counts only when the specified process is running. According to the results presented in the previous section, however, both scenarios allow to classify websites with success rates of over 80% for SVM. This is because the test systems were mostly idling during the experiments. If this is not the case, the effects of increased system load can be countered by increasing the profiling time in order to obtain more information from the renderer process. Furthermore, an adversary can train multiple models for multiple levels of system load to account for an unknown load of the target system. Also, a slight increase of the number of guesses yields a significant increase in classification success. These measures also help against other impairments of the measurement quality, e.g., if multiple websites are opened in parallel in the Tor browser, if direct URLs are used instead of visiting the homepage of a website, if the number of profiled websites grows, or if websites are visited that

have not been profiled and cannot be classified as a consequence.

Compared to Google Chrome in Incognito mode, the results of the Tor Browser are worse on average. This can be explained with the browser start-up phase, which is always captured for Tor. Also, random network delays introduce jitter in the observations of the website loading. Large delays require to prolong the event acquisition phase, otherwise the success rates are expected to drop. Another adverse effect is the changing geo-location of the Tor exit nodes. Many websites, particularly news sites like New York Times and Yahoo, customize their appearance based on the location of their visitors and therefore introduce additional noise in the measurements. Similar effects can occur if websites contain personalized advertisements and other frequently changing content. While this potentially decreases the success rates, we believe that an important part of the profiling relies on the website templates.

Among the Machine Learning techniques, Convolutional Neural Networks have proven to be the most capable for classifying websites, if enough samples are available. This is the reason why CNNs performed well in Google Chrome and Tor Browser experiments, but not in ARM experiments. CNNs are built for multi-classification of complex structures by extracting meaningful features. On the contrary, SVM and kNN are designed to create hyperplanes to separate space into classes. Since the number of dimensions is high in the experiments, it is difficult to find the best hyperplane for each dimension. Nevertheless, there is still a need for further studies on CNN, since the results could be improved by modifying the parameters, number of layers and neurons.

In general, the feasibility of website fingerprinting via hardware performance events is not limited to the specific profiling scenarios and test platforms used in our experiments. This is because of the fundamental phenomenon that loading

different websites creates different microarchitectural footprints. This a logical consequence of optimized software that is designed to provide best user experience. Therefore, similar results are expected also for other x86 and ARM processors, as well as for other HPE interfaces and web browsers, unless mitigation strategies are implemented.

7.5 Outcome

When websites are loaded in the browser, they stress the underlying hardware in a distinct pattern that is closely related to the contents of the website. This pattern is reflected in the microarchitectural state of the processor that executes the browser, which can be observed with high precision by counting hardware performance events. Since these events can be legitimately measured by user space applications, it is feasible to infer opened websites via performance event measurements. We showed this by utilizing machine learning techniques, achieving recognition rates of up to 99% with 5 guess in Incognito mode. In addition, the results show that CNN is more powerful to obtain better classification rates from high number of classes in the presence of noise. By applying CNN, the whistleblowing websites are classified with 79% accuracy among 40 websites while the overall classification rate increases up to 89.25% with 5 guesses in Tor browser.

Chapter 8

Machine Learning based Side-Channel Attacks on Mobile Platforms

8.1 Motivation

Today, more than 1 billion people use Android applications [151]. The security and privacy of these applications are therefore of great relevance. The Android OS therefore employs a variety of protection mechanisms. Apps run in sandboxes, inter-process communication is regulated, and users have some degree of control via the permission system. The majority of these features protects against software-based attacks and logical side-channel attacks. The processor hardware, however, also constitutes an attack surface. In particular, the shared processor cache heavily speeds up the execution of applications. As a side effect, each application leaves a footprint in the cache that can be profiled by others. These footprints, in turn, contain sensitive information about the application activity. Jana et al. [94] showed

that browsing activity yields unique memory footprints that allow to infer accessed web pages. Oren et al. [127] demonstrated that these footprints can be observed in the cache even from JavaScript code distributed by a malicious website. While these attacks have succeeded based on a solid amount of engineering, the increasing complexity of applications, operating systems (OS), and processors make their implementation laborious and cumbersome. Yet, studying side-channel attacks is important to protect security and privacy critical applications in the long term. We believe that machine learning techniques, especially Deep Learning (DL), can help make side-channel analysis significantly more scalable. DL thereby reduces the human effort by efficiently extracting relevant information from noisy and complex side-channel observations. At the same time, DL introduces a new risk as attacks become more potent and easier to implement in practice.

In this work, we demonstrate this risk and compile a malicious Android application, which, despite having no privileges or permissions, can infer private user activities across application and OS boundaries. With the App, we are able to detect other running applications with high confidence. With this information, we focus on activities that happen within an application. We detect visited websites in Google Chrome and identify videos that are streamed in the Netflix and Youtube applications. Those inferences are possible by analyzing simple last-level cache (LLC) observations of at most 6 seconds with advanced DL algorithms. The entire attack succeeds in well under a minute and reveals sensitive information about the mobile phone user. None of the currently employed protection mechanisms prevent our attack, as the LLC is shared between different processes and can be monitored from user space. Our cache profiling technique is based on the Prime+Probe (P+P) attack [152], which relies on cache eviction to monitor certain cache sets. In contrast to previous work, we implement this eviction with a dynamic eviction set test that

succeeds even for imprecise timing sources, random line replacement policies, and missing physical memory addresses information. This comes at the cost of measurement accuracy and introduces a certain amount of noise in the cache observations. We counter this effect by applying machine learning to the observations, and compare classical algorithms to advanced deep learning along the way. While Support Vector Machines (SVMs) and Stacked AutoEncoders (SAEs) struggle during the classification, Convolution Neural Networks (CNNs), a DL technique, succeed in efficiently extracting distinct features and classifying the observations. As CNNs have recently gained attention in the field of side-channel analysis, we explain our parameter selection and compare it to related work. For the implementation of our attack, we neither require the target phone to be rooted nor the malicious application to have certain privileges or permissions. On our test device, a Nexus 5X, the Android OS is up-to-date and all security patches are installed. The malicious code runs in the background, requires no human contribution during the attack, and draws little attention due to the short profiling phase.

Our Contribution. In summary, we

- propose an inference attack on mobile devices that works without privileges, permissions, or access to special interfaces.
- find eviction sets with a novel dynamic timing test that works even with imprecise timing sources, random line replacement policies, and virtual addresses only.
- classify cache observations using ML/DL techniques (SVMs, SAEs, CNNs) and thereby infer running applications, opened websites, and streaming videos.
- achieve classification rates up to 98% with a profiling phase of at most 6

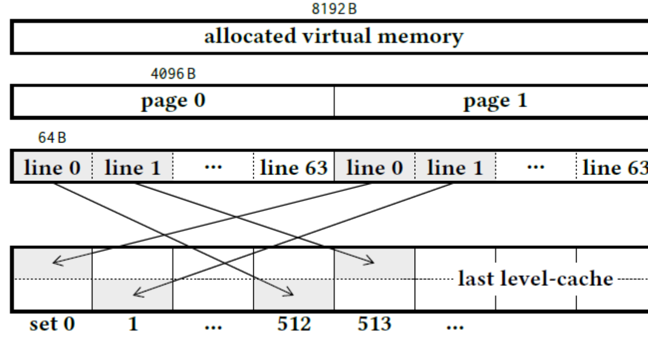


Figure 8.1: Mapping of virtual memory to cache sets.

seconds. The entire attack succeeds in well under one minute.

8.2 Inference Attack

The threat model of our proposed inference attack assumes that a mobile device user installs a malicious application from an app store on Android. This happens regularly, as malicious apps offer benign functionality to disguise malicious background activities (e.g. hidden crypto currency mining). The malicious code needed for our attack operates from user space and does not need any app permissions. This means that we neither require a rooted phone, nor ask the user for certain permissions, nor rely on any exploits, e.g., to escalate privileges or to break out of sandboxes. Furthermore, we do not rely on features or programming interfaces that might not be available on all Android versions. The sole task of our malicious code is to profile the LLC and classify victim activities with pre-trained ML/DL models. Once the LLC profiles have been gathered, the models are queried to infer sensitive information.

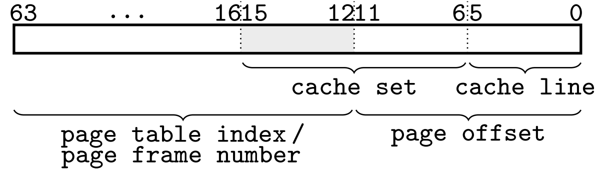


Figure 8.2: Virtual/physical address and its interpretation.

8.2.1 Attack Outline

The proposed inference attack consists of two main phases. In the **training phase**, the attacker creates ML/DL models on a training device that is similar to the target device. Ideally, the processor and operating system are identical on both devices. The models are created by recording raw LLC profiles of target applications, websites, and videos, followed by preparing the feature vectors, and training the ML/DL algorithms with them. The trained models are then directly integrated into the malicious application, which is subsequently published in the app store. In the **attack phase**, the malicious app prepares eviction sets for profiling the LLC on the target device. Subsequently, the LLC sets are profiled in a Prime+Probe manner and the feature vectors are extracted. Finally, the feature vectors are classified with the pre-trained models to infer opened applications, visited websites, and streamed videos. All steps of the attack phase are lightweight and can be executed in the background without drawing notable attention.

8.2.2 Finding Eviction Sets

Once deployed, the first task of the malicious app is to find eviction sets on the target device. An eviction set is a group of memory addresses that map to the same cache set. These addresses are called *set-congruent*. Figure 8.1 illustrates the problem of finding set-congruent addresses and forming eviction sets. The graphic shows a block of virtual memory that is backed by two fixed-size memory pages. In

the figure, we assume a common page size of 4 KiB. As soon as an address within the block is accessed, the corresponding memory content is brought into the processor cache. Since the cache manages data on fixed-size cache lines, one access will cache multiple bytes. We assume a common cache line size of 64 bytes. The illustrated cache is set-associative and holds multiple lines per cache set. Memory that is brought into the cache is deterministically assigned to a cache set. For last-level caches, this assignment is commonly derived from physical addresses that are unavailable to most user space applications. Figure 8.2 illustrates the link between virtual and physical addresses, and how they are interpreted by the cache. The most significant bits of each virtual address are the page table index, while the least significant bits are the page offset. A page size of 4 KiB yields $log_2(4096) = 12$ offset bits. The page table index is used to lookup an entry in the page tables that contains the page frame number. The page frame number and the page offset form the physical address. The page offset bits are thereby identical in both virtual and physical address. The least significant bits of the physical address are then used by the cache controller for basic indexing. The lowest bits are used to address a byte on a cache line, while the subsequent bits determine the cache set in which the address will be placed. For 1024 cache sets, 10 bits are used as cache set index. As shown in Figure 8.2, these bits do not fit entirely within the page offset (highlighted in gray). Therefore, the exact cache set cannot be determined from the virtual address, as the most significant index bits are unknown. This complicates the mapping of virtual addresses to cache sets and may cause consecutive pages to map to completely different parts of the cache, as indicated in Figure 8.1. Finding eviction sets from user space is therefore a non-trivial problem. To fully solve it, one must *(a)* group virtual addresses according to cache sets, and *(b)* obtain the correct order such that group 0 maps to cache set 0 and so on. We refer to this as an *ordered* mapping

Algorithm 3 Finding eviction sets.

```
1: 𝒯 = {}
2: t_old = 0
3: for i from 1 to n do
4:     add(𝒯, i)
5:     t_𝒯 = access(𝒯, r)
6:     if (t_𝒯 − t_old) > τ_jump then
7:         ℰ ← {}
8:         for all p in 𝒯 do
9:             t_p = access(𝒯n{p}, r)
10:            if (t_𝒯 − t_p) > τ_jump then
11:                add(ℰ, p)
12:            end if
13:        end for
14:        report(ℰ)
15:        remove(𝒯, ℰ)
16:        t_old = access(𝒯, r)
17:    else
18:        t_old = t_𝒯
19:    end if
20: end for
```

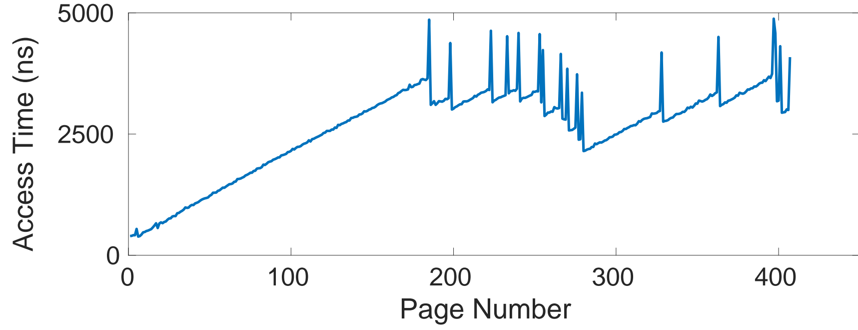
between virtual addresses and cache sets. In practice, one must obtain physical address bits to derive this mapping, e.g., by gaining elevated privileges [108] or by exploiting additional vulnerabilities [62]. Alternatively, it is possible to find an *unordered* mapping that only fulfills *(a)*. This can be achieved with search algorithms that perform simple timing measurements and thereby find groups of set-congruent virtual addresses. While the algorithms do not reveal which group corresponds to which cache set, they can be run entirely from user space. In literature, the study by Vila et al. [160] investigates this type of search algorithms. The authors give a comprehensive overview of previous approaches, but limit their evaluation to Intel processors. We discuss approaches relevant to this work in the following paragraph and refer the interested reader to this study for further information.

Lipp et al. [108] compile eviction sets from physical addresses which they obtain

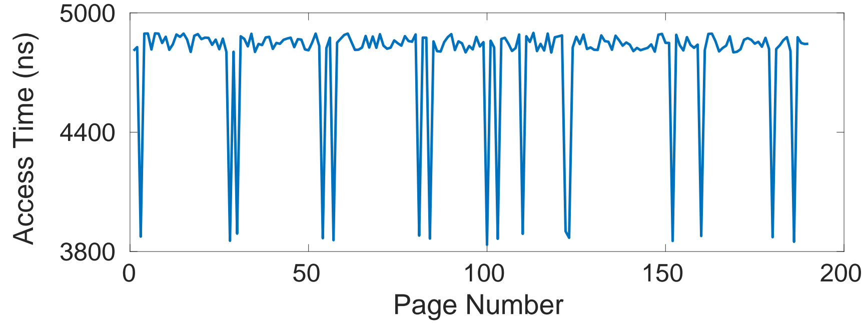
from the *pagemap* file that is present on many Linux systems. After their work was published, Android restricted the access to *pagemap* entries from user space.[1] Irazoqui et al. [86] and Gruss et al. [61] rely on *huge pages*, i.e. pages with typical sizes of bigger than 1 MiB. Huge pages increase the page offset and thereby reveal the missing bits that determine the cache set. Oren et al. [127] and Bosman et al. [29] rely on special page allocation mechanisms in web browsers and operating systems that simplify the eviction set search. Genkin et al. [51] build eviction sets from sandboxed code within a web browser, while only relying on virtual addresses. Yet, they still require a precise and low-noise timing source to distinguish cache hit and miss. In contrast to these previous works, we propose a search algorithm that neither relies on physical addresses (whether obtained from pagemap, huge pages, or elsewhere), nor on certain features of memory allocators, nor on a precise timing source. Our approach for finding eviction sets is purely based on virtual addresses and robust against imprecise and noisy timing sources. In addition, we found our approach to be resilient against the random line replacement policy implemented in many ARM application processors.

Algorithm 3 outlines our approach for finding eviction sets. Prior to execution, we assume that n memory pages have been requested and are available as a memory pool. Note that we do not pose any requirements on the memory pages, thus, our algorithm works with any page size, including, but not limited to, 4 KiB. Since we want to evict the entire LLC, we need to choose n such that the requested memory area is large enough to fill it. In our experiments, we request a memory area that is twice as large as the LLC. This turned out to be sufficient for deriving all eviction sets. Algorithm 3 iterates over the allocated memory pages in sequential order. Each unused page is first added to a temporary eviction set $\mathcal{T}$ (line 4). Next, the

[1] `https://source.android.com/security/bulletin/2016-03-01`



(a) Average page access time, $t_{\mathcal{T}}$, for an increasing number of pages in $\mathcal{T}$.



(b) Average page access time, t_p, used to filter an eviction set from $\mathcal{T}$.

Figure 8.3: Plots of (a) $t_{\mathcal{T}}$ and (b) t_p, as used in Algorithm 3.

first byte of each page in $\mathcal{T}$ is accessed and the average time $t_{\mathcal{T}}$ of this access cycle is measured. The parameter r determines how often the access cycle is repeated. In each cycle, all pages in $\mathcal{T}$ are accessed once. The overall timing is then divided by r to obtain the average. This is useful to account for imprecise timing sources and different replacement policies. A detailed discussion of r is given later in this section. The access time $t_{\mathcal{T}}$ is then compared with the time from the previous loop cycle (line 6), where $\mathcal{T}$ was one page smaller. If the time difference is higher than a threshold τ_{jump}, then there is a systematic contention in a cache set. In other words, the pages in $\mathcal{T}$ entirely fill one cache set and cause a line replacement in the process. This is illustrated in Figure 8.3(a), which shows the average access time $t_{\mathcal{T}}$ over an increasing number of pages in $\mathcal{T}$. As long as no set contention occurs, the average timings increase steadily. Once a contention happens, the timing peaks. Each peak

in the plot indicates that one cache set is completely filled.

After a set contention is detected, Algorithm 3 iterates over all pages in $\mathcal{T}$, temporarily excludes one of them from $\mathcal{T}$, accesses this reduced set, and stores the average access time in t_p (line 9). If the time difference between $t_{\mathcal{T}}$ and t_p is again bigger than τ_{jump}, then the excluded page p belongs to the eviction set. This is illustrated in Figure 8.3(b), which shows the average access time t_p for all candidate pages p in $\mathcal{T}$. As soon as a candidate is part of the eviction set, the systematic set contention vanishes and the access time t_p drops. Each drop in the plot therefore indicates a member of the eviction set. Those pages are then added to the final eviction set $\mathcal{E}$, which is reported on line 14. The entries in $\mathcal{E}$ are subsequently removed from $\mathcal{T}$, before the outer loop continues to add unused pages to $\mathcal{T}$. Once the outer loop reaches n, the reported eviction sets are expanded. This procedure is outlined in the following paragraph.

Eviction Set Expansion and Duplicates. Each of the m eviction sets reported by Algorithm 3 contains a list of memory pages. Since one page fits more than one cache line, we can derive multiple evictions sets from one $\mathcal{E}$. This is indicated in Figure 8.1. With 4 KiB pages and 64-byte cache lines, there are 64 lines on one page. As memory pages are contiguous physical memory, we know that those 64 lines belong to 64 consecutive cache sets. Hence, we can derive 64 adjacent eviction sets from one $\mathcal{E}$ (the first being $\mathcal{E}$ itself) by simply adding multiples of 64 to the start address of the pages. Depending on how large we chose n, it can happen that Algorithm 3 reports more than one eviction set for each cache set. We therefore need to check all reported eviction sets for duplicates and remove them. This procedure is outlined in Algorithm 4. It starts by storing the indices of all m reported eviction sets from Algorithm 3 in the list $\mathcal{B}$ for bookkeeping (line 2). As long as $\mathcal{B}$ is not empty, the first index is removed and assigned to f (line 4). The algorithm then

Algorithm 4 Removing duplicates.

```
1:  F = {}
2:  B = {1..m}
3:  while notempty(B) do
4:      f = pop(B)
5:      for s in B do
6:          t_E = access([onepage(E_f), E_s], r)
7:          if (t_E > τ_jump) then
8:              remove(B, s)
9:          end if
10:     end for
11:     add(F, f)
12: end while
```

iterates over all remaining indices s and accesses the corresponding eviction sets $\mathcal{E}_f$ and $\mathcal{E}_s$. In particular, one page in $\mathcal{E}_f$ and all pages in $\mathcal{E}_s$ are accessed consecutively, and the whole process is repeated r times. If the average timing $t_{\mathcal{E}}$ of these access cycles is larger than a threshold τ_{jump}, then $\mathcal{E}_f$ and $\mathcal{E}_s$ map to the same cache set. If this happens, the affected index s is removed from $\mathcal{B}$ (line 8), and the iteration continues. After all eviction sets have been tested, the index in f is added to the final list $\mathcal{F}$ (line 11). With each loop, $\mathcal{B}$ is shrinking as duplicate eviction sets are removed. Once $\mathcal{B}$ is empty, $\mathcal{F}$ contains the list of unique eviction set indices.

Timer Precision and Noise. Both algorithms 3 and 4 are designed to compensate imprecise and noisy timing sources. Although previous works [51, 139] suggest that accurate timers can be crafted even in environments that restrict access to high-precision timing sources, this engineering effort can be saved here. We believe this adds to the practicality of our attack. The precision and noise compensation in our algorithms is done by tuning the parameter r, as well as the threshold τ_{jump}. r defines the number of access cycles, i.e., how often a selection of memory pages is accessed. τ_{jump} defines how the timings of these access cycles are evaluated. In Algorithm 3, the accesses on line 5 will typically cause cache hits until the gathered

pages trigger a systematic set contention. The difference between $t_{\mathcal{T}}$ and t_{old} will therefore be in the order of t_{miss}, where t_{miss} is the duration of a cache miss. Similarly, the accesses on line 9 will cause cache hits, if the candidate page p is part of the eviction set. In this case, the difference between $t_{\mathcal{T}}$ and t_p will again be around t_{miss}. Therefore, τ_{jump} can initially be set slightly smaller than t_{miss}. Adjustments can be made subsequently based on experimental data. The choice of r depends on the precision of the timer and the measurement quality. In general, r should be set such that $r \cdot t_{miss}$ is larger than the precision of the timer. It can be increased further, if high levels of noise are encountered, e.g., due to high system load. The choices for τ_{jump} and r also hold for Algorithm 4, where the accesses on line 6 will typically cause cache hits until $\mathcal{E}_f$ and $\mathcal{E}_s$ are duplicates. When this happens, a systematic set contention will occur, as the chosen page from $\mathcal{E}_f$ will be evicted by $\mathcal{E}_s$. In our experiments, we set r between 900 and 1000, and τ_{jump} to 500. The timer available on our test device, a Nexus 5X, provides a precision of 52 ns. This corresponds to approximately 100 clock cycles. In many related attacks (e.g. [173]), where timers typically have clock cycle accuracy, this rather low resolution would already introduce difficulties. In our approach, we simply tune r to compensate the low resolution.

Line Replacement Policies. We can also use r to compensate the effects of replacement policies. This is because the parameter r causes repetitive accesses to cache lines, which signals the cache controller that the accessed lines are of heightened interest and should not be replaced. For least-recently-used (LRU) policies, this is obviously beneficial, as unrelated cache activity will less likely interfere with the eviction set finding. But also random replacement policies benefit, because averaging over r access cycles attenuates the effect of unintended line replacements that happen due to random line selection. Our experiments outlined in Section 9.2

show that the choice of r as stated above is sufficient to compensate the effects of the random line replacement found on our test device.

Implementation and Limitations. Only few requirements have to be satisfied to find eviction sets with our approach. Memory must be allocated and accessed, and the accesses must be timed. Memory pages can be of arbitrary size and the timing source can be coarse-grained. This allows our algorithm to be implemented in user space and, thus, in a plethora of environments beyond mobile devices, e.g., desktop computers and cloud servers. Even sandboxes and virtual machines are typically no obstacle, enabling remote attacks, e.g., from JavaScript. The limitation of our approach is that the exact mapping of eviction sets to cache sets remains unknown. However, this is not a deficiency but a direct consequence of not knowing physical addresses. This choice makes our approach more practical and, thanks to the application of machine learning, still allows successful inference attacks.

Performance. We evaluated Algorithms 3 and 4 on our test device with the parameters stated previously. The targeted last-level cache is 16-way set-associative and contains 1024 cache sets. It implements a random replacement policy and features 64-byte cache lines. Requested memory pages have a standard size of 4 KiB. Based on 1000 evaluation runs, Algorithms 3 and 4 successfully yield all eviction sets with an average runtime of 20 seconds. Note that during our inference attack, eviction sets must be found only once and remain unchanged until the malicious app is restarted.

8.2.3 Post-processing and Feature Vectors

With the eviction sets obtained from Algorithms 3 and 4, the last-level cache can be profiled in a traditional Prime+Probe [152] manner. This is done by filling each

cache set with the corresponding eviction set (prime), before re-filling it immediately afterwards (probe). High levels of activity in the cache set will increase the probing time, whereas low levels will keep it low. Each probing time constitutes a *measurement sample* of the cache set. In our experiments, we measure n_T samples per cache set. The overall activity profile of the last-level cache, in short *LLC profile*, consists of the samples of all cache sets. Before these samples are classified by the machine learning algorithms, they are post-processed and converted to feature vectors. The following list outlines the post-processing steps.

1. Elimination of timing outliers with a threshold τ_O. All outliers are replaced with the sample median. In our experiments, τ_O is set to $5\,\mu$s .

2. Conversion of timing samples to binary representation. A threshold τ_H decides whether a sample value is *low* or *high*. In our experiments, τ_H is set to 750 ns.

3. Sample compression by grouping high samples. Bursts of consecutive high samples are reduced to a single high value.

The removal of outliers reduces noise in the measurements. The binary representation simplifies interfacing with the machine learning algorithms and distills cache activity to two categories: *high* and *low*. Sample compression reduces data complexity while keeping the essential information of whether there was high or low activity. It also alleviates the effect of random replacement policies, as it compensates self-eviction during the probe step. From the post-processed measurement samples we derive three different feature vectors that are outlined in the following paragraphs.

Unordered Feature Vector. This feature vector is called *unordered*, because the exact mapping of eviction sets to cache sets is unknown, as explained in Section 8.2.2.

This means that it is unclear which region of the cache a given sample stems from. However, it is possible to determine which region of the memory page a sample belongs to, because the addresses in a given eviction set share a common page offset. We leverage this observation to further compress the feature vector and reduce training complexity. In particular, we sum up the high samples of all cache sets that belong to the same page offset. With 4 KiB pages and 64-byte cache lines there are 64 distinct page offsets. Thus, the final feature vector contains 64 values.

FFT Feature Vector. For this feature vector, the post-processed measurement samples are converted with a fast Fourier transformation (FFT). The purpose of the FFT is to further reduce measurement noise, which has been pointed out by previous work [66, 127]. The FFT turns the n_T time-domain samples per cache set into $\frac{n_T}{2}$ frequency components (excluding the DC component). The employed sampling rate n_S is derived by dividing 1 second by the duration of one Prime+Probe cycle. We reduce the complexity of the $\frac{n_T}{2}$ frequency components by compressing them to n_F final components with a sliding window. These final components are again summed up over all cache sets that belong to the same page offset. Thus, the final feature vector contains $n_F \cdot 64$ values.

8.3 Experiment Setup and Results

In total, we conduct three experiments in which our malicious app detects running applications, visited websites, and streamed videos. The attack targets are given in Appendix A.2. For each machine learning algorithm, we build a multi-class classifier. 90% of the measured LLC profiles are thereby selected randomly for the training phase, while the rest of the data is chosen to evaluate the efficiency of the trained classifier. This 10% holdout approach yields the classification rates that

are presented in this section. The rates are thereby based on the most likely label. Throughout the experiments, we observed that 10-fold cross-validation results are consistent with a 10% holdout approach. We also evaluated our classifiers against *unknown* inputs accounting for activity the models have not been trained with. In total, we collected more than 800 GB of cache profiling data to evaluate our inference attack.

8.3.1 Target Device

We use a Google Nexus 5X with Android v8.0.0 for our experiments. It features four ARM Cortex-A53 and two ARM Cortex-A57 processor cores. The malicious code runs on one of the A57 cores and profiles the LLC in the background. The LLC on the A57 core cluster contains 1024 cache sets. The target applications are launched and transition automatically to the A57 processor cluster. This is because the scheduler assigns resource-hungry processes (e.g. browser or multimedia applications) to the A57 cores to leverage their high performance. During all experiments, the system was connected to the campus wireless network and background processes from the Android OS and other apps were running. The timing source in our malicious app is the POSIX *clock_gettime* system call, which is available on all Android versions as part of the Bionic standard C library [21]. For website inference, we run Google Chrome and for video inference, we run the Netflix and YouTube apps.[2]

8.3.2 ML/DL Configuration

The following paragraphs discuss the parameter selection of the machine learning algorithms and provide further details about their usage. SVM and SAE classification is implemented with the help of LibSVM [34], whereas CNN classification is

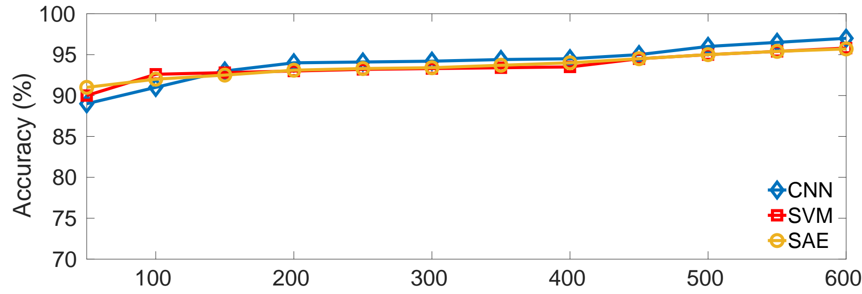
[2]Chrome v64.0.3282.137, Netflix v6.16.0, YouTube v13.36.50.

done using custom Keras [37] scripts together with the Tensorflow [15] GPU backend. The CNN is trained on a workstation with two Nvidia 1080Ti (Pascal) GPUs, a 20-core Intel i7-7900X CPU, and 64 GB of RAM.

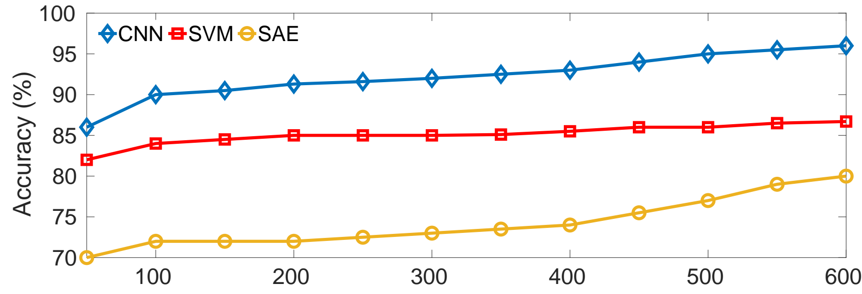
SVM. The *ordered* and *unordered* feature vectors are classified with a linear SVM, while a non-linear SVM is used for the *FFT* feature vector. This is because the FFT is computed with non-linear functions (*cos*, *sin*) and the labels are linearly increasing for the classes. This choice is verified in preliminary experiments. Similarly, we determine that the linear kernel type outperforms radial basis and polynomial options for the *unordered* and *FFT* feature vectors.

SAE. The SAE is constructed with two hidden layers of 250 and 50 neurons, respectively. The maximum number of epochs is set to 400, since no improvements can be observed afterwards. We decrease the effect of over-fitting by setting the L2 weight regularization parameter to 0.01. The output layer is a softmax layer.

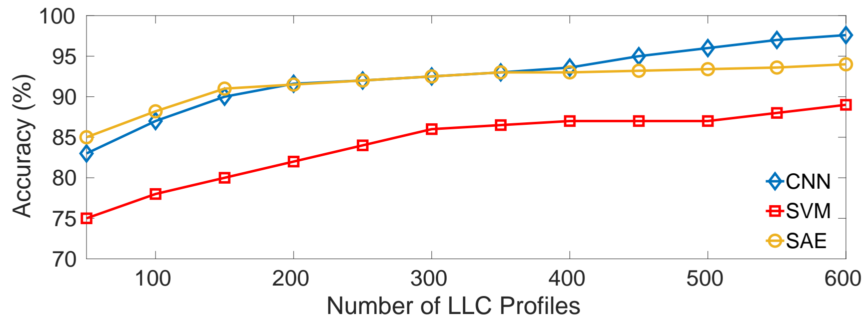
CNN. The CNN consists of two 1-D convolution layers that are followed by maxpooling, dropout, as well as flatten and dense layers. The selection of the layer parameters is done with the help of preliminary experiments. Table A.5 in Appendix A.2.2 shows the parameter space that we explored. Eventually, we selected the parameters that yielded the lowest validation loss (highlighted in bold). The size of the first 1-D convolution layer is varied from 8 to 1024. The lowest validation loss is obtained with a size of 512. Similarly, the size of the second convolution layer is varied between 32 and 256, and eventually fixed to 256. A third convolution layer does not improve classification. The activation function in the convolution layers is set to rectified linear unit (ReLU). The size of the subsequent maxpooling layer is varied from 2 to 8. The default size of 2 yields the best results. The dropout of the following dropout layer is varied between 0.1 and 0.5, and finally set to 0.2. A higher



(a) Ordered.



(b) Unordered.



(c) FFT.

Figure 8.4: Classification results for application inference over an increasing number of LLC profiles for (a) *ordered*, (b) *unordered*, and (c) *FFT* feature vectors.

dropout, as for example used in computer vision, adversely affects the classification. Next, the kernel size is adjusted and, out of the values between 3 and 27, a size of 9 achieves the lowest validation loss. A flatten layer shapes the data in our network, before a dense layer with size 200 and *tanh* activation function is appended. Finally, we employ a set of standard choices: the kernel initializers are chosen uniformly at random, an Adam optimizer is used to speed up the training phase, and the batch size is set to 50, as its effect on the classification rate is negligible.

8.3.3 Evaluation Results

The following sections present the evaluation results for application, website, and video inference.

8.3.3.1 Application Inference

For this attack, we target 100 random mobile applications from the Google Play Store, including dating, political, and spy apps. The full list is given in Table A.2 in Appendix A.2. The first 70 apps are used to train and evaluate the machine learning models, while the remaining 30 apps are treated as being unknown. Each app is started and profiled for 1.5 seconds as described in Section 8.2. Within this time frame, we collect $n_T = 1,500$ measurement samples per cache set. For the FFT computation, the sampling rate $n_S = 1.9\,MHz$ and the number of bins $n_F = 15$. A comparison of the machine learning techniques and feature vectors is given in Figure 8.4. It contains three sub-plots that each show the classification results of SVM, SAE, and CNN over an increasing number of recorded LLC profiles. Recall that 90% of the recorded profiles are used for training, whereas the rest is used to obtain the classification rates shown in the plots. The stated numbers of LLC profiles only reflect the measurement effort for the training phase, which is done offline on a training device. In the attack phase on the target device, recording a single LLC profile is sufficient to conduct a successful inference attack. The same holds for the results shown in figures 8.8 and 8.9. Plot 8.4(a) illustrates the results of the comparison attack, which is based on the *ordered* feature vector. The ordered profiling allows all three classifiers to distinguish applications with high confidence. CNN even achieves a classification rate of 97%. Plot 8.4(b) shows that classification rates drop, if the LLC profiles are based on the *unordered* feature vector. SAE even falls down to 80%, while CNN remains above 95%. The CNN we designed is

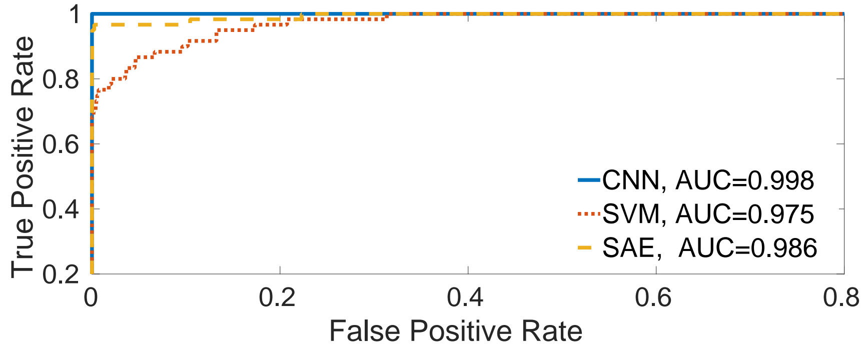


Figure 8.5: Average receiver operating characteristic (ROC) curves for SVM, SAE, CNN during application inference.

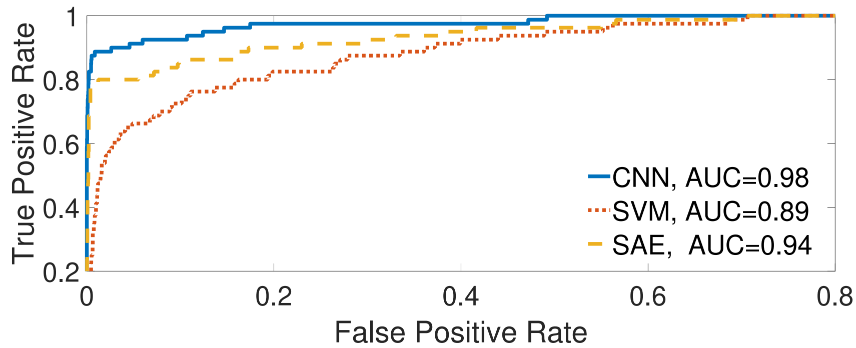


Figure 8.6: Average receiver operating characteristic (ROC) curves for SVM, SAE, CNN during website inference.

therefore least affected by the unknown mapping between eviction and cache sets. As shown in Plot 8.4(c), the classification rates improve again, if the *FFT* feature vector is used. In particular, CNN and SAE benefit from this transformation, while SVM cannot fully leverage the information in the frequency spectrum. Our CNN reaches a classification rate of 97.8% and is thereby able to fully close the gap to the comparison attack.

A further performance metric for the three machine learning techniques is shown in Figure 8.5. It displays the receiver operating characteristic (ROC) curves for the *FFT* feature vector. For multi-class classification, ROC curves are computed for each class against all remaining classes ($1 \times$ N-1). The final ROC curve is then the average over all computed ROC curves. Figure 8.5 also provides the area under the

curve (AUC) values in the plot legend. The higher the AUC, the less the machine learning technique suffers from false positives. While all three classifiers produce low false positive rates, CNN outperforms SVM and SAE. Based on the results of the application detection, we conclude that the CNN is the most suitable classifier for our inference attack. For website and video inference, we will therefore only present the results of the CNN.

Unknown Applications. The inherent nature of supervised learning is to recognize events that are similar to those used in the training phase. In practice, however, events may occur that the model has never been trained with. This also applies to our inference attack. Naturally, we cannot train our models with all existing applications on the app store. In fact, we want to focus only on apps that are of interest. Hence, we need a way to recognize and filter apps we have not trained yet. We achieve this by monitoring the probability estimates obtained from the softmax layer. Recall that we train only 70 apps out of the 100 that are given in Table A.2. When we classify all 100 apps on our target device, we obtain the probability estimates shown in Figure 8.7. All known apps yield a high probability estimate close to 1, whereas unknown apps yield estimates that are significantly lower. We thus label each classification that yields a probability estimate below a threshold to be *unknown*. This threshold can be tuned according to attack requirements. A low value ensures that no application is missed during the attack. However, this leads to the detection of apps that have not been executed (false positives). A high threshold increases the confidence that all detected apps have actually been running. However, this comes at the cost of misclassifying known apps to be unknown (false negatives). As a general rule, we recommend to set the threshold at the intersection of the probability distributions obtained from the softmax layer. In our experiments, we chose a threshold of 0.84, which is illustrated as a dashed line in Figure 8.7. With

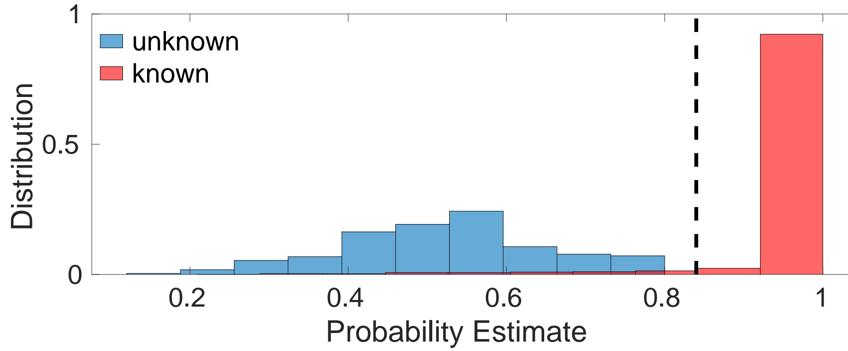


Figure 8.7: Probability estimates from the CNN softmax layer while classifying known and unknown apps.

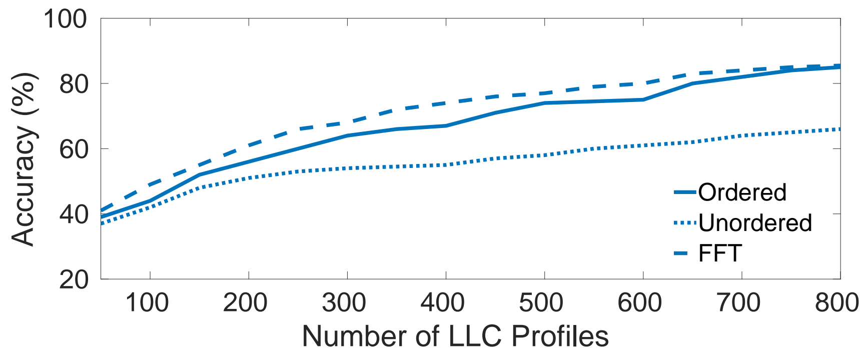


Figure 8.8: Website classification with our CNN for *ordered* (solid), *unordered* (dotted), and *FFT* (dashed) feature vectors.

this approach, our inference attack works reliably even in the presence of unknown applications on the target device.

8.3.3.2 Website Inference

The results in the previous section illustrate that our malicious app can reliably detect running applications with high confidence. Once a browser is detected, the app tries to infer websites that are currently viewed. For this attack, we target 100 different websites that are visited in Google Chrome. The list of websites is given in Table A.1 in Appendix A.2. To emphasize that browsing histories are sensitive information, the list includes news, social media, political, and dating websites. For each website, we profile the LLC for 1.5 seconds and again obtain $n_T = 1,500$

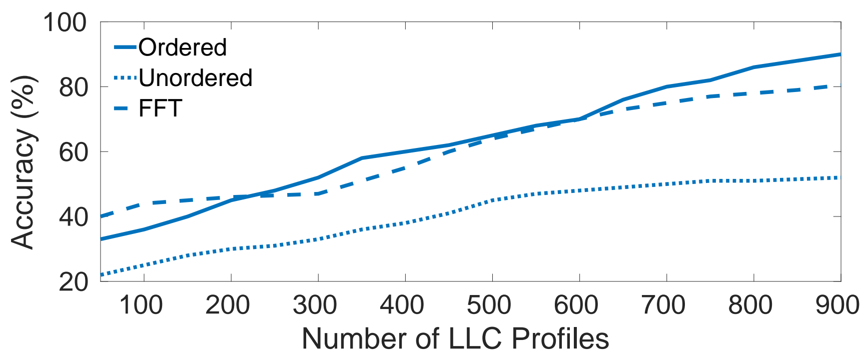


Figure 8.9: Video classification with our CNN for *ordered* (solid), *unordered* (dotted), and *FFT* (dashed) feature vectors.

samples per cache set. The features vectors are constructed in the same way as for application detection. Figure 8.8 shows the CNN classification results for all three feature vectors over an increasing number of LLC profiles. Similar to application inference, the *FFT* results match and slightly overshoot the results of the comparison attack. With a classification rate of 86%, the CNN is able to infer viewed websites with satisfactory confidence. The classification rate is lower compared to application inference, because loading and rendering websites leaves a weaker footprint in the last-level cache than opening apps. The ROC curves for the *FFT* feature vector are shown in Figure 8.6. The AUC values in the plot legend again illustrate that our CNN yields the lowest number of false positives. The CNN classifier and the *FFT* feature vector are therefore the best choices for website inference.

Unknown Websites. As previously, we train the CNN with only 70 websites and subsequently classify all 100 websites from Table A.1. The probability estimates of the softmax layer are similar to the application inference and are thus not shown for the sake of brevity.

8.3.3.3 Video Inference

Similar to website inference, our malicious app also tries to detect videos that are being streamed in the Netflix and YouTube applications. We therefore target a total of 20 videos, which are given in Table A.4 in Appendix A.2. In contrast to previous evaluations, we increase the profiling phase to 6 seconds. This is because the LLC footprint of videos is significantly less distinct compared to applications and websites. Within the extended profiling phase, we collect $n_T = 6,000$ samples per cache set. For the FFT computation, the number of bins, n_F, is increased to 60. Due to the high number of feature values, the size of the first convolution layer in our CNN is increased to 1024. The rest of the feature vectors are constructed in the same way as for application and website inference.

Figure 8.9 shows the CNN classification results for all three feature vectors over an increasing number of LLC profiles. The *FFT* results again match the comparison attack, but eventually fall behind by 10%. With a classification rate of 80%, our inference attack is able to infer streaming videos with moderate success. We believe that the LLC profiles do not contain enough information to distinguish multiple videos, as video processing is a rather homogeneous task. In addition, parts of the video decoding are typically outsourced to the GPU, which further reduces the cache footprint. Regarding the ROC curves, which are shown in Figure 8.10, the CNN again outperforms SVM and SAE. Due to the reduced success rate for video classification, we skipped the evaluation of unknown videos. Yet, we expect it to follow the same trend as for application and website inference.

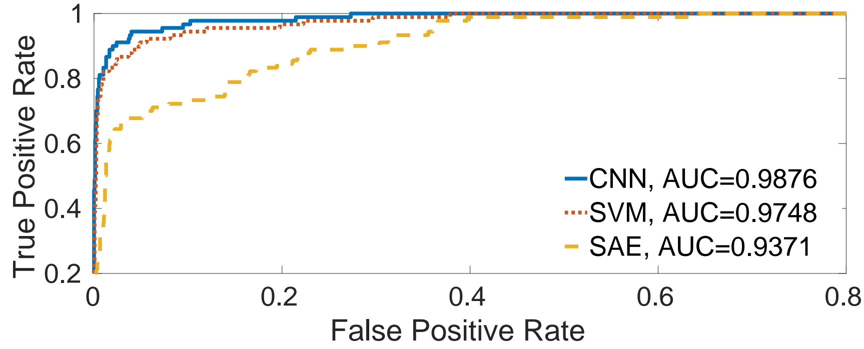


Figure 8.10: Average receiver operating characteristic (ROC) curves for SVM, SAE, CNN during video inference.

8.4 Discussion

The previous section shows that modern machine learning techniques enable successful inference attacks even when simple cache profiling methods are employed. Throughout our experiments, frequency-domain transforms of LLC profiles yield high success rates when being classified by a CNN. The FFT thereby reduces the noise in the measurements, while the CNN distills consistent features despite the lacking order with which the cache sets are profiled. The resulting classification closely matches the comparison attack that is based on precisely ordered LLC profiles obtained with the help of additional attack steps. Clearly, an adversary can omit these steps when using our inference attack. The limit of our attack becomes apparent when the LLC activity is less distinct or faints. While applications and websites can reliably be inferred, the accuracy drops for video classification. This could be improved by increasing the profiling time or including other side-channels such as GPU activity. Nevertheless, the results clearly indicate that a carefully crafted and well-trained CNN enables inference attacks that are robust, easy to implement, and therefore practical.

Attack in Numbers. The pre-trained CNN model is approximately 24 MiB large. Together with the attack code, this yields a total app size of 25 MiB. The other ML

models are significantly smaller. If the number of target classes increases, the size of the models grows linearly. When the app is launched, it first creates the eviction sets required for LLC profiling. As stated in Section 8.2.2, this takes 20 seconds on average. Recording one LLC profile takes at most 6 seconds. The subsequent classification is also a matter of seconds. The work by Ignatov et al. [79] is a useful reference to assess the performance of CNN classification on Android phones. On the Nexus 5X, all CNN classification benchmarks finish in under 13 seconds, yielding a total attack time of well under a minute.

Attack Portability. Our inference attack is not limited to the device and scenario presented in this work. The attack components are flexible and can be ported easily. The *eviction set algorithms* presented in Section 8.2.2 are generic and can be adapted to other environments with appropriate choices of r and τ_{jump}. The algorithms are robust against changes in cache size, number of sets, associativity, and replacement policy. They will therefore find eviction sets not only on ARM-based mobile devices but also on x86 systems. With the *unordered* and *FFT* feature vectors introduced in Section 8.2.3, the exact mapping of eviction sets to cache sets is not required for an attack. This has multiple advantages. First, the attack does not require physical addresses and can be launched entirely from user space. Second, it is agnostic to the page size of the system and works with pages from less than 4 KiB to multiple MiB. Third, the attack can be launched without additional and complex attack steps (e.g. [62]) that would increase the attack effort and lower the practicality. The designed *convolutional neural network* is a suitable fit for the resulting cache observations. It allows to distill the cache footprint of virtually any activity occurring on the target system. Future research may study detecting exact versions of applications or input events such as swipes, touches, or the like. The CNN presented in this work is a good starting point for any new attack scenario.

Since a fine-tuning of the parameters may be necessary, Table A.5 can be consulted for sensible parameter ranges. In summary, our inference attack is versatile and constitutes a threat not only to mobile applications, but also to virtual machines and containers on servers and any desktop software. We consider the exploration of other attack scenarios as future work.

Countermeasures. The inference attack proposed in this work has two fundamental requirements. First, it relies on the mutual eviction of cache lines from the last-level cache. This eviction can be impaired by cache flushing [54], cache partitioning [110], scheduling [157] and line replacement policies [93]. However, most of these approaches require changes to the processor hardware or introduce substantial performance overhead. The second requirement is to time memory accesses. While disabling access to timers or overlaying them with noise [77] complicates attacks, these strategies seem far from sustainable, as timing sources can be crafted artificially even in restrictive execution environments [139, 51]. Another approach is to craft adversarial examples against DL based classification models [56]. Inci et al. [80] recently showed that CNN-based side-channel attacks can be prevented by adding specially crafted noise to performance counters. This approach could also be adopted by applications running on mobile devices. A general defense strategy is the detection of ongoing attacks, e.g., by monitoring the memory access behavior of programs [179]. However, most detection approaches are probabilistic and, thus, suffer from false positives and false negatives. The generic nature of our inference attack renders it extremely difficult to defend against, especially without dedicated support from the processor hardware.

8.5 Outcome

Inference attacks undermine our privacy by revealing our most secret interests, preferences, and attitudes. Unfortunately, modern processors, which constitute the core of our digital infrastructure, are particularly vulnerable to these attacks. Footprints in the processor cache allow the inference of running applications, visited websites, and streaming videos. Above all, the advances in machine learning, especially the concepts behind deep learning, significantly lower the bar of successfully implementing inference attacks. Our work demonstrates that it is possible to execute an inference attack without privileges, permissions, or access to special programming interfaces and peripherals. The simple nature of the attack code makes a comprehensive defense extremely difficult. This simplicity is paired with the careful application of deep learning. Interferences such as measurement noise, misalignment, or unfavorable processor features are thereby conveniently compensated. The comparison with concurrent work furthermore indicates that inference attacks of this kind are ubiquitous and succeed across runtime environments and processing hardware. For applications that value the privacy of their users, protection against inference attacks is therefore of utmost importance. A comprehensive solution, however, seems to require a closer collaboration between hardware manufacturers, operating system designers, and application developers.

Chapter 9

FortuneTeller: Machine Learning based Defense Mechanism

9.1 Motivation

In the past decade, we have witnessed the evolution of microarchitectural side-channel attacks [172, 60], from being considered as a nuisance and largely dismissed by chip manufacturers to becoming front-page news. The severity of the threat was demonstrated by the Spectre [97] and Meltdown [109] attacks, which allow a user with minimal access right to easily read arbitrary locations in the memory by exploiting the transient effect of illegal instruction sequences. This was followed by a plethora of attacks [154, 153, 138] either extending the scope of the microarchitectural flaws or identifying new leakage sources. It is noteworthy that these critical vulnerabilities managed to stay hidden for decades. Only after years of experimentation, researchers managed to gain sufficient insight into, for the most part, the unpublished aspects of these platforms. This leads to the point that they could formulate fairly simple but very subtle attacks to recover internal secrets. Therefore,

the natural question becomes: how can we discover dormant vulnerabilities and protect against such subtle attacks? A fundamental approach is to eliminate the leakage by using formal analysis. However, given the tremendous level of complexity of modern computing platforms and lack of public documentation, formal analysis of the hardware seems impractical in the near future. What remains is the modus operandi: **leaks are patched as they are discovered by researchers through inspection and statistical analysis**.

Countermeasures for microarchitectural side-channel attacks focus on the operating system (OS) hardening [110, 59], software synthesis [32] and analysis [166, 167], and static [87] or dynamic [177, 35, 30] detection of attacks. Static analysis is performed by evaluating the untrusted software against known malicious code patterns without running it on a target platform [87]. Alternatively, dynamic analysis aims to detect malicious behaviors in the system by analyzing the runtime footprint of the running processes [35]. Existing works on dynamic detection of microarchitectural attacks are based on collecting footprints from the hardware performance counters (HPCs) and limited modeling of malicious behaviors [35, 123, 177, 30]. A crucial challenge for both detection techniques is the shortage of knowledge about new attack vectors. Therefore, modeling malicious behaviors for undiscovered attacks and accurately distinguishing them from benign activities are open problems. Moreover, microarchitectural attacks are in infancy and supervised learning models, which are used as attack classifier [123], are not reliable to detect known attacks due to the insufficient amount and imprecise labeling of the data. Hence, unsupervised methods are more promising to adapt the detection models to real-world scenarios.

Anomaly-based attack detection, which has been also studied in other security applications [142, 45], aims to address the aforementioned challenge by only modeling the benign behaviors and detecting outliers. While there have been several efforts

on anomaly-based detection of cache attacks [30, 35], modern microarchitectures have a diverse set of components that suffers from side-channel attacks [174, 60, 122]. Thus, detection techniques will not be practical and usable, if they do not cover a broad range of both known and unseen attacks. This requires more advanced learning algorithms to comprehensively model the entire behavior of the microarchitecture. On the other hand, statistical methods for anomaly detection are not sufficient to analyze millions of events that are collected from a very complex system like the modern microarchitecture. A major limitation of the classical statistical learning methods is that they use a hand-picked set of features, which wastes the valuable information to characterize the benign execution patterns. As a result, these techniques fail at building a generic model for real-world systems.

The latest advancements in Deep Learning, especially in Recurrent Neural Networks (RNNs), show that time-dependent tasks such as language modeling [147], speech recognition [136] can be learned and upcoming sequences are predicted more efficiently by training millions of data samples. Similarly, computer programs are translated to instructions where the corresponding microarchitectural events have time-dependent behaviors. Modeling the sequential flow of these events for benign applications is extremely difficult by using logic and formal reasoning due to the complexity of the modern microarchitecture features. We claim that these time-dependent behaviors can be modeled on a large scale by observing a sufficient number of benign execution flows since the long-term dependencies in the time domain can be learned with a high accuracy by training Long-short term memory (LSTM) and Gated Recurrent Unit (GRU) networks. Besides, a challenging task of detecting the correlations in benign applications are done automatically by LSTM/GRU networks in the training phase without any expert input.

Our Contribution: We propose *FortuneTeller* which is the first generic detec-

tion model/technique for microarchitectural attacks. *FortuneTeller* learns the benign behavior of hardware/software systems by observing microarchitectural events, and detects any outlier that does not conform to the trained model as malicious behavior. *FortuneTeller* is also able to detect unseen microarchitectural attacks since the tool only requires training over benign execution patterns.

In summary, we propose *FortuneTeller* which:

- captures the system-wide low-level microarchitectural traces and learn noisy time-dependent sequences through advanced RNN algorithms by training a more advanced and generic model.
- can detect malicious behavior dynamically in an unsupervised manner including stealthy cache attacks (Flush+Flush), transient execution attacks (Meltdown, Spectre, Zombieload) and Rowhammer.
- performs better by comparing it to the state-of-the-art detection techniques.

9.1.1 Methodology

Our conceptual design for *FortuneTeller* consists of offline and online phases as shown in Figure 9.1: In the offline phase, *FortuneTeller* collects time sequence data from diverse set of benign applications by monitoring security sensors in the system. The collected dataset is used as the training data and it is fed into the RNN algorithm with a sliding window technique. The weights of the trained model are optimized by the algorithm itself since each data sample is also used as the validation. When there is no further improvement in the validation error, the training process stops. Once the RNN model is trained, it is ready to be used in a real-time system.

In the online phase, the real-time sequences are captured from the same security sensors and given as input to RNN model. The prediction of the next measurement

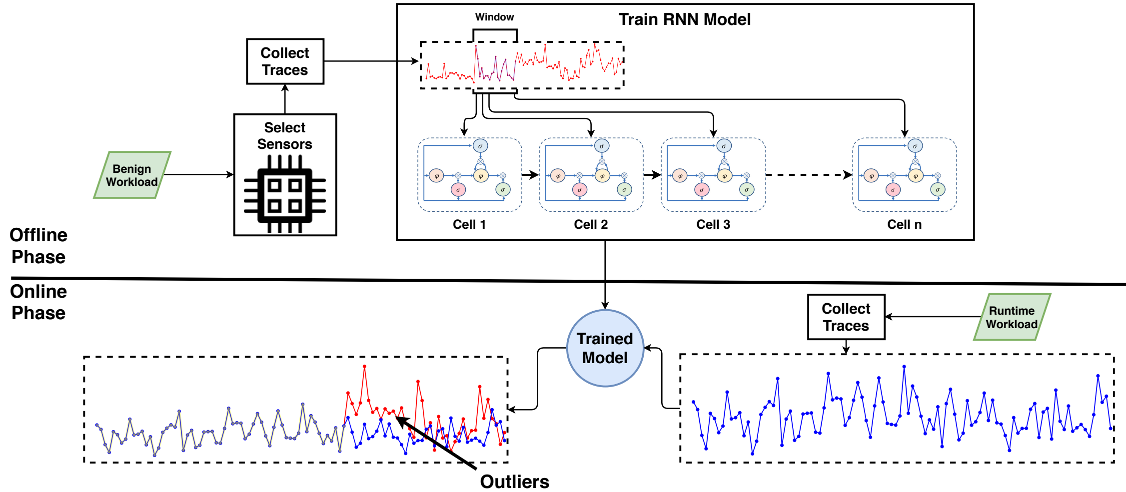


Figure 9.1: *FortuneTeller* implementation

for each sensor is made by the pre-trained RNN model, dynamically. If the mean squared error (MSE) between the predicted values and real-time measurements is consistently higher than the previously determined threshold, the anomaly flag is set. The online phase is the actual evaluation of *FortuneTeller* in a real-world system.

Two separate detection models are trained with LSTM and GRU networks since they are known for their extraordinary capabilities in learning the long time sequences. Our purpose is to train an RNN-based detection model, which can predict the microarchitectural events of benign executions in the next time steps with minimal error. In our detection scenario, we consider a time series $X = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}\}$, where each measurement $\mathbf{x}^{(t)} \in R^m$ is an m-dimensional vector $\{x_1^{(t)}, x_2^{(t)}, \ldots, x_m^{(t)}\}$ and each element corresponds to a sensor value at time t. As all temporal relations can not be discovered from millions of samples, a sliding window with a size of W is used to partition the data into small chunks. Thereby, the input to RNN algorithm at time step t is $\{x_1^{(t-W+1)}, x_2^{(t-W+2)}, \ldots, x_m^{(t)}\}$, where the output is $\mathbf{y}^{(t)} = \mathbf{x}^{(t+1)}$. Note that, even though there is a fixed length sliding window in the problem formulation, the overall input size is not fixed. Finally, the trained model is saved to be used in real-world system.

To evaluate the performance of the RNN model, a new dataset is collected from both benign applications and attack executions. This dataset is used as a test dataset which is fed into the model to calculate the prediction error in the next time steps. The error at time step $t+1$ is $e^{(t+1)}$ which is equal to $1/m \sum_{i=1}^{m}(y_i^{t+1} - x_i^{t+1})^2$. The model predicts the value of the next measurement and then, the error for each counter is summed up and normalized.

To detect the anomalies in the system, a decision window D and an anomaly threshold τ_A are used. If all the predictions in D are higher than τ_A, the anomaly flag F_A is set in the system in Equation 9.1.

$$F_A = \begin{cases} 1, & \forall e^{(t+1)} \in D \geq \tau_A \\ 0, & otherwise \end{cases} \tag{9.1}$$

The choice of D directly determines the anomaly detection time. If D is chosen as a small value, the attacks with a very small footprint can be caught. On the other hand, the false alarm risk increases in parallel, which is controlled by adjusting τ_A. This trade-off is discussed further in section 9.2.

9.1.2 Implementation

9.1.2.1 Profiled Benchmarks and Attacks

The main purpose of *FortuneTeller* is to train a generic model by profiling a diverse set of benign applications. Therefore, selecting benign applications is of utmost importance. The benign application dataset is collected from benchmark tests in Phoronix benchmark suite [10] since the suite includes different types of applications such as cryptographic implementations, floating-point, and compression tests, web-server workloads, etc. The complete list is given in Appendix, Table A.7. It

is important to note that some benchmark tests have multiple sub-tests and all the sub-tests are included in both training and test phases. In addition to CPU benchmarks, we evaluate our detection models against system, disk and memory test benchmarks. To increase the diversity, the daily applications such as web browsing, video rendering, Apache server, MySQL database and Office applications with several parameters are profiled for real-world examples.

The benign applications are divided into training and test applications. The first 67 tests in Table A.7 are monitored to collect training dataset while the remaining applications are profiled to obtain test dataset which is used to compute the Sensitivity, Specificity and F-score of the RNN models. In our work, false positives represent the number of times a benign application is classified as an attack/anomaly by the model in one minute.

For the attack executions, we include cache attacks such as F+F, F+R, and P+P attacks. Different from previous works, these attacks are applied on arbitrary memory blocks to avoid any assumption on the target implementation. Additionally, Spectre (v1 and v2) and Meltdown are implemented to read secrets such as passwords in a pre-determined memory location. Also, two types of Rowhammer attacks namely, one-sided and double-sided, are applied to have bit flips. To test the efficiency of *FortuneTeller* we implemented a recent microarchitectural attack, ZombieLoad, to steal data across processes. For this purpose, a victim thread leaks pre-determined ASCII characters and the attacker reads the line-fill buffer to recover the secret. If the alarm flag is set during the execution of the attack, it is true positive (TP). On the other hand, the undetected attack execution is represented by false negative (FN).

9.1.2.2 Performance Counter Selection

In our detection model, we leverage HPCs as security sensors. Although the number of available counters in a processor is greater than 100, it is not feasible to monitor all counters concurrently. In an ideal system, we should be able to collect data from a diverse set of events to be able to train a generic model. However, due to the limited number of concurrently monitored events, we choose the most optimum subset of counters that give us information about common attacks. In our experiments, we leverage Intel PCM tool [84] to capture the system-wide traces. This tool is generic to all Intel processors so that it can be used in any Intel processor regardless of its family/version. The set of counters in our experiments is chosen from *core* counters. The main reason to choose core counters is the high variety of the available counters such as branches, cache, TLB, etc. In total, 36 counters were tested in the selection phase, which is listed in Appendix, Table A.6.

In the data collection step, a subset of the counters is profiled concurrently, which is limited to three or four in Intel processors. For each subset, a separate training dataset from 30 different Phoronix benchmark tests [10] (1-30 in the Table A.7) is collected until all 36 counters are covered. We also collect a test dataset from 20 benchmark tests (31-50 in the Table A.7) and 6 microarchitectural attacks (Attacks: 1-6 in the Table A.7). The Zombieload attack is not included in the performance counter selection phase since it was not released at that time. The sampling rate is chosen as 10 ms to cause minimal overhead in the system.

For every subset of counters one LSTM model is trained, where four-dimensional data is given as an input to the LSTM model and then, the final counters are selected based on their F-score given in Appendix, Table A.6. Branch-related counters have a higher correlation for Meltdown and Spectre attacks. However, the F-score is also around 0.3 as real-world applications also use the branches heavily. It is important

to note that since speculative branches are commonly integrated into the benign applications, branch counters are not useful to detect speculative execution attacks in the wild. Thanks to our LSTM based counter selection technique, finding the most valuable counters is fully automated and the false alarm rate is decreased significantly.

The first selected counter is $L1_Inst_Miss$ which is successful to detect Rowhammer, Spectre and Meltdown attacks with F-score of 0.8023. As a second counter, $L1_Inst_Hit$ is chosen, since Flush+Flush and Flush+Reload attacks are detected with a high accuracy and the F-score is 0.8137. The reason behind the high F-score is that the *flush* instruction is heavily used in those attacks and the instruction cache usage also increases in parallel. Interestingly, Flush+Flush attack is known as a stealthy microarchitectural attack however, it is possible to detect it by monitoring instruction cache hit counter. The last selected counter is LLC_Miss, which allows to detect Rowhammer and Prime+Probe attacks with high accuracy. These attacks cause frequent cache evictions in the LLC, which increases the number of anomalies in the LLC_Miss counter. Even though it is allowed to monitor up to 4 counters on the Intel server systems like Xeon, we selected 3 counters to profile for anomaly detection. The reason behind this is that in the desktop processors (Intel Core i5, i7) the programmable counters are limited to 3. The individual counter experiments show that the individual counters are not efficient to detect all the microarchitectural attacks. Therefore, there is a need for the integration of the aforementioned 3 counters to detect all the attacks with a high confidence rate.

9.2 Evaluation

In this section, we explain the experiments which are conducted to evaluate *FortuneTeller*. The experiments aim to answer the following research questions: 1) How does *FortuneTeller* perform in predicting the next performance counter values for benign applications with the increasing number of measurements (Subsection 9.2.2)? What is the lowest possible FPR for server (Subsection 9.2.3) and laptop environments (Subsection 9.2.4)? 3) How does the size of the sliding window affect the performance of *FortuneTeller* (Subsection 9.2.5)? 4) How realistic is real-time protection with *FortuneTeller* (Subsection 9.2.6)? 5) How much performance overhead is caused by *FortuneTeller* (Subsection 9.2.7)?

9.2.1 Experiment Setup

FortuneTeller is tested on two separate systems. The first system runs on an Intel Xeon E5-2640v3, which is a common processor used on server machines. It has 8 cores with 2.6 GHz base frequency and 20 MB LLC. The second device is used to illustrate a typical laptop/desktop machine, which is based on Intel(R) Core(TM) i7-8650U CPU with 1.90 GHz frequency. It has 8MB LLC and 2 cores in total.

Two types of RNN models, LSTM and GRU, are used to train *FortuneTeller*. The sliding window size, batch size and number of hidden LSTM/GRU layers are kept the same in the training phase. Training of RNN models is done using custom Keras scripts together with the Tensorflow and GPU backend. The models are trained on a workstation with two Nvidia 1080Ti (Pascal) GPUs, a 20-core Intel i7-7900X CPU, and 64 GB of RAM.

9.2.2 RNN Model Training

The first step in building *FortuneTeller* is to learn the pattern of the benign applications. This is not an easy task since the chosen benchmarks and real-world applications have complex fingerprint in the microarchitectural level due to the system noise. Moreover, the fingerprint at each run is not identical and the execution of the application takes at least several seconds, which makes the learning the long-term correlations challenging. To model the patterns of benign executions, we use one LSTM block which consist of 100 cells. The model has a loss function as MSE and adam optimizer is used to update the weights with a batch size of 1.

10 random benchmarks are chosen, and a separate RNN model for each of them is trained. The validation error obtained as a result of training is the critical metric to determine the capacity of the RNN algorithms as it indicates how well *FortuneTeller* guesses the next counter value. The initial RNN model is trained with only 1 measurement and the number of measurements is increased gradually up to 44. It is observed that there is no further improvement in the validation error after 36 measurements for both LSTM and GRU networks in Figure 9.2. Note that, the training data is scaled to [0 1] and the validation error is the average error of the 3 counters.

The prediction trend of $ICache.Hit$ counter is obtained by training the LSTM model as shown in Figure 9.3. The solid line represents the actual counter value while two other lines show the prediction values. When LSTM model is trained with one measurement, the predictions are not close to actual value. It means the model could not optimize the network weights to learn the pattern. On the other hand, once the number of measurements is increased to 36, the predictions become more consistent. Increasing the number of measurements would increase the efficiency of the model slightly but it also directly affects the training time of the model. If the

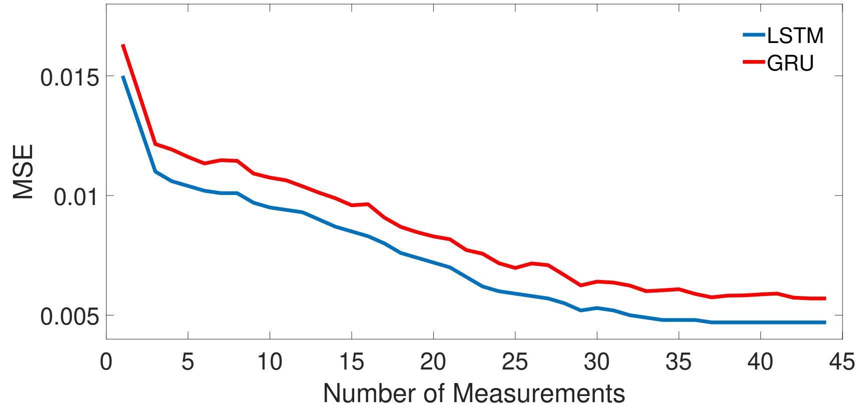


Figure 9.2: Validation error with increasing number of measurements for Gnupg benchmark

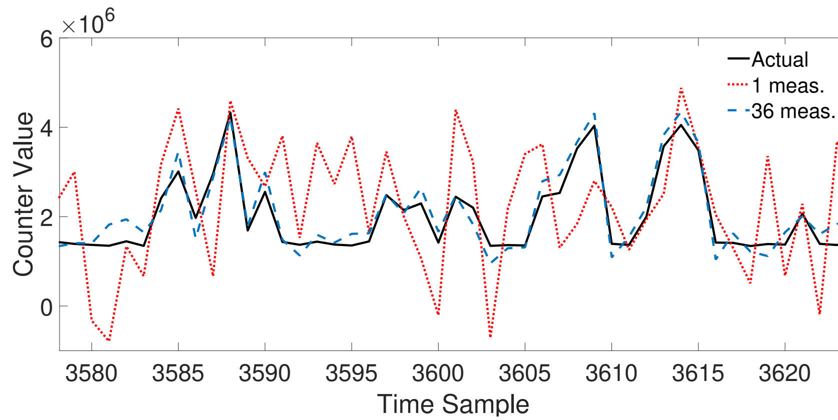


Figure 9.3: Prediction error in Gnupg for LSTM algorithm

dataset is unnecessarily huge, the training time increases linearly. Therefore, it is decided to collect 36 measurements from each application in the training phase of *FortuneTeller* to achieve the best outcome from RNN algorithms in the real systems. With accurate modeling of the benign behavior, the number of false alarms is reduced significantly. This is the main advantage of *FortuneTeller*, since the previous detection mechanisms apply a simple threshold technique to detect the anomalies when a counter value exceeds the threshold. In contrast, *FortuneTeller* can predict sudden increases in the counter values and thus, correct classification is performed more efficiently than before.

9.2.3 Server Experiments

The first set of experiments is conducted in the server machine. Three core counters, $ICache.Miss$, $ICache.Hit$ and $LLC.Miss$ are monitored concurrently in the data collection phase. The training dataset is collected with a sampling rate of 10 ms from 3 core counters during the execution of benign applications. The dataset has 4 million samples in total, collected from 67 randomly selected benchmark tests as listed in Appendix, Table A.7. Firstly, the LSTM model is trained with the collected dataset where the input size is $3 \times 4,000,000$. The sliding window size is selected as $W = 100$, which means the total number of LSTM units is 100. A full analysis of the effect of different window sizes is given in Subsection 9.2.5. The training is stopped after 10 epochs since the validation error does not improve further. The validation error decreases to 0.0015. The training time for 4,000,000 samples takes approximately 2 days.

After the LSTM model is trained, a new dataset for the test phase is collected from counters by profiling remaining 106 benign benchmark tests (not included in training phase), 100 random websites rendering in Google Chrome, MySQL, Apache, Office applications and microarchitectural attacks. The number of samples obtained from each application changes between 1000-20000. Each application is monitored 50 times, and then, the test data is fed into the LSTM model to predict the counter values at the next time steps. Moreover, to make the test phase more realistic, the number of applications running concurrently is increased up to 5. The applications are chosen randomly from the test list in Appendix Table A.7, and started at the same time. 100 measurements are collected from concurrently running applications. In total, 20 million samples are collected for the test phase.

The threshold τ_A and decision window D are chosen to obtain a low value for FPR while increasing the TPR in parallel. For the lower τ_A values, the sensi-

tivity increases but the number of false alarms increase in parallel. Therefore, we decided to increase τ_A until we achieve an acceptable FPR. Once τ_A reaches 2.6×10^6, *FortuneTeller* has 0.90 sensitivity and the corresponding decision window is $D = 30$ samples. Additionally, the FPR stays around 0.05% (specificity = 0.9995) per minute which means every 2,000 minutes (33 hours) there is a chance of false alarm. Since it is more important to keep the false alarm rate lower for real-world deployment, the parameters are chosen accordingly. With the decreasing τ_A values, the number of true positives begins increasing in parallel with the false alarm rate. Therefore, the decision window and threshold values can be chosen based on the desired sensitivity of the detector.

The results indicate that P+P attack is the most difficult attack to be detected by the LSTM model in the server. This result is expected since P+P attack mostly focuses on specific cache sets and the cache miss ratio is smaller than other types of attacks. Besides, the instruction cache is not heavily used by P+P attack, which makes the detection more difficult for *FortuneTeller* due to the lack of strong fingerprint. On the other hand, the highest TPR is obtained for Flush+Reload and Rowhammer attacks with 100% and 0% FNR. As these attacks increase the number of data cache misses and instruction hits through the extensive use of *clflush* instruction, the fluctuation in the counter values is higher than for the other types of attacks and benign applications. The accuracy of predicting the next values decreases when the variance is high in the counters, thus, the prediction error increases in parallel. Since the higher prediction rate is a strong indicator of the attack executions in the system, *FortuneTeller* detects them with a high accuracy. Note that, ZombieLoad is also detectable by the *FortuneTeller*, even though it was not included in the performance counter selection phase. This shows that *FortuneTeller* can even detect previously unknown microarchitectural attacks with the current trained models.

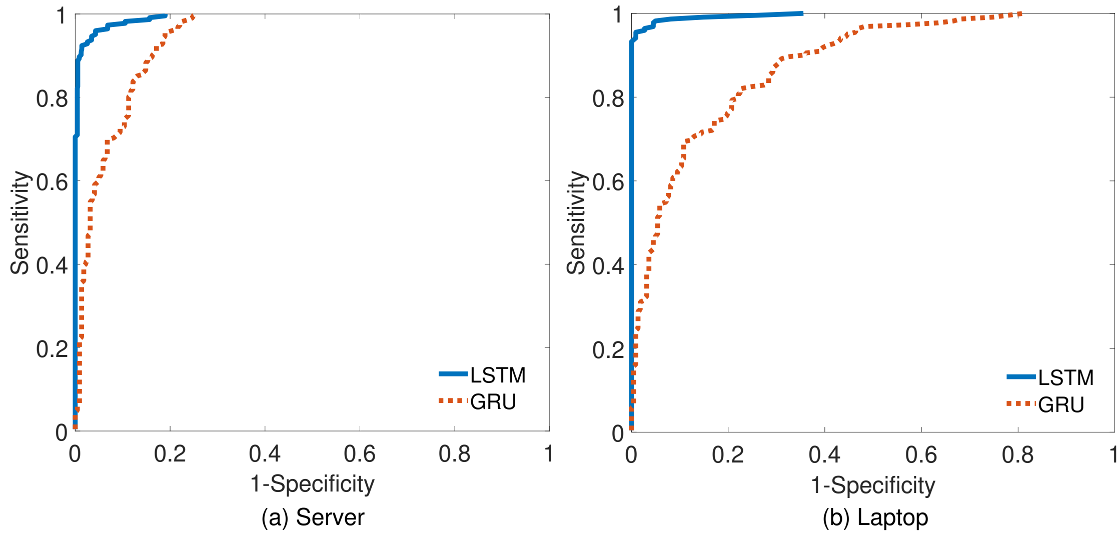


Figure 9.4: ROC curve for LSTM and GRU models in server and laptop environments

The ROC curves in Figure 9.4a demonstrate that LSTM networks are better than GRU networks to detect the anomalies. The counter values are predicted with a higher error rate in GRU networks, which makes anomaly detection harder. Some benign applications are always detected as an anomaly by GRU, thus, the FPR is always high for different threshold values. The AUC (Area Under the Curve) for LSTM model is very close to perfect classifier with a value of 0.9929. On the other hand, AUC for GRU model is 0.9395, which is significantly worse than the LSTM model. There are several reasons behind the poor performance of GRU networks. The first reason is that GRU networks are not successful to learn the patterns of Apache server applications since there is a high fluctuation in the counter values. Also, when the number of concurrently running applications increases, the false alarms increase drastically. Nevertheless, LSTM networks are good at predicting the combination of patterns in the system. Therefore, the FPR is very low for the LSTM model.

9.2.4 Laptop Environment

The experiments are repeated for the laptop environment to evaluate the usage of *FortuneTeller*. LSTM and GRU models are trained with 4 million samples, which are collected from benign applications. Since laptops are mostly used for daily work, the counter values are relatively smaller than in the server scenario. However, the applications stress the system more than the server scenario due to the few (four) physical cores.

When we analyze the relation between D and τ_A, we observe the same situation as in the server scenario. The lower τ_A values are not sufficient to distinguish the anomalies from benign executions. Therefore, we need to choose the optimal τ_A value slightly higher than in the server scenario, with a value of 3.8×10^6. The corresponding D value is 60, which means that the anomalies are detected in 600 ms. The sensitivity improves up to 95% while specificity stays around 99.98. The decision window is 300 ms longer than the server scenario however, the performance of *FortuneTeller* is better in the laptop scenario. In Figure 9.4b, the ROC curves of LSTM and GRU models are compared. The AUC value of the LSTM model is considerably higher than the GRU model with a value of 0.9970. However, the AUC value for GRU is 0.8985. This shows that LSTM outperforms GRU model to predict the counter values of benign applications. Hence, FNR and FPR are lower for LSTM models.

Among the attack executions, the Rowhammer attack can be detected with 100% success rate since the resulting prediction error of the RNNs is very high. The other attacks have similar prediction errors, hence, *FortuneTeller* can detect the attacks with the same success rate. Since the computational power of laptop devices is low, the concurrently running applications have more noise on the counter values. Therefore, the prediction of the counter values is more difficult for RNN algorithms.

Table 9.1: The False Alarm Rate in percentage per second for applications

	Server (%)		**Laptop (%)**	
	LSTM	GRU	LSTM	GRU
Benchmarks	0.18	0.24	0.06	0.08
Websites	0.05	0.06	0.02	0.15
Videos	0.00	0.00	0.00	0.06
MySQL	0.00	0.00	0.00	0.00
Apache	0.00	0.13	0.00	0.11
Office	0.00	0.00	0.00	0.00
2 Apps	0.00	0.00	0.00	0.08
3 Apps	0.03	0.10	0.00	0.09
4 Apps	0.08	0.22	0.04	0.12
5 Apps	0.13	0.36	0.06	0.15

While LSTM networks have small FPR for 4 and 5 applications running at the same time, GRU networks are not efficient to classify them as benign applications.

The overall results show that LSTM works better than GRU networks for both laptop and server scenarios, as shown in Table 9.1. The first and second values represent the LSTM and GRU false alarm rates per minute in percentages, respectively. In the server scenario, videos, MySQL and Office applications never give false positives. Websites running in Google Chrome trigger few false alarms. Therefore, the FPR is around 0.02% per minute for LSTM network in the server scenario overall. The main disadvantage of GRU networks is the poor performance in the prediction when the number of applications increases. The FPR and FNR are approximately 0.12%. This shows that the number of false alarms is 6 times more for GRU based *FortuneTeller*.

9.2.5 Optimizing the Sliding Window Size

We analyze the effect of the sliding window size on anomaly detection with 10 ms sampling rate in the server environment. 12 different window sizes are used to train LSTM and GRU models. The window size starts from 25 and is increased by 25 at

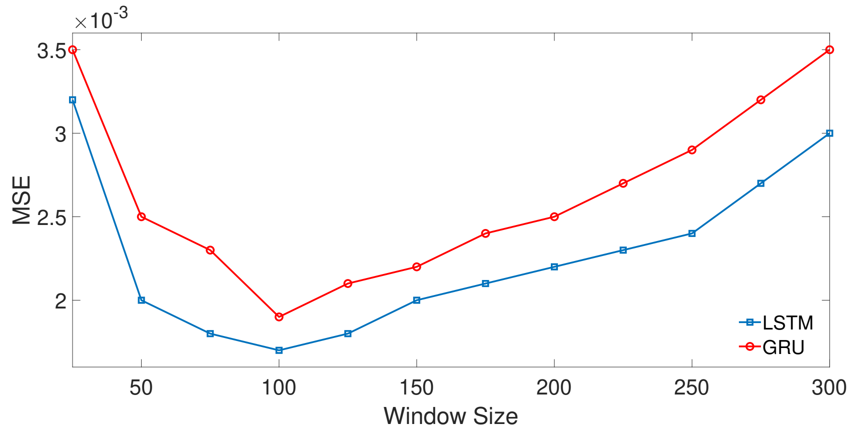


Figure 9.5: The validation error for different sizes of the sliding window

each step until reaching 300.

The changes in the validation error for both LSTM and GRU networks are depicted in Figure 9.5. The overall GRU training error is higher than the LSTM network for each window size. Both models reach the lowest error when the sliding window size is 100. Even though LSTM and GRU are designed to learn long sequences, it is recommended to choose the window size between 50-150. The increase in error for larger window sizes shows that the statistical dependency of performance counter values weakens when the trained sequence is longer. Therefore, independent values in the large window sections cause a raise in the validation error. In other words, the ability of the trained model on guessing the next counter values gets worse. Therefore, we recommend to use a window size of 100, as all the models in the previous experiments are trained with this parameter. It is also important to note that the training time increases proportionally to the size of the window, which increases the training time unnecessarily window sizes greater than 100.

9.2.6 Prediction Time in Testing Phase

The dynamic anomaly detection is heavily affected by the time it takes to predict the next counter values. Thus, the sampling interval needs to be longer than the

computation time of the next predicted value. In our experiments, we observed that the prediction time is proportional to the size of the model. Since GRU has fewer cells in the architecture, the prediction from GRU model is received faster. The LSTM outputs the prediction values within 5 ms while GRU needs 4 ms. Even though GRU is 20% faster than LSTM in the prediction phase, due to the high FPR of GRU networks, *FortuneTeller* is trained with LSTM networks. Additionally, to decrease the performance overhead of the prediction process, the prediction is made every 10 ms.

9.2.7 Performance Overhead

The performance overhead of *FortuneTeller* is less than a typical prevention overhead [112]. In the server environment, the overhead value is obtained with a sampling rate of 10 ms for data collection and the predictions from the LSTM model are received in parallel from GPU cores. We achieved a 1.8% overhead in average while CPU and GPU have 1% and 3% overhead, respectively. The overhead of individual benchmark tests fluctuates between 0.1% and 10% for benign applications. If the GPU cores are disabled, the CPU has 3.5% overhead in total for data collection and prediction.In the laptop scenario, the number of cores is lower than the server scenario and there is no dedicated GPU core to make the predictions. When the sampling and prediction interval are 10 ms, the overhead is 2.4%, which is applicable in real-time setup. This overhead is also lower than in the server scenario, if there is no GPU.

9.3 Comparison of *FortuneTeller* with Prior Detection Methods

Several studies have focused on microarchitectural attack detection. While some works [30, 35, 177] use unsupervised techniques, Mushtaq et al. [123] benefits from supervised ML methods. All proposed methods claim that the false positive rate is very low in real-world scenarios. However, all these detection techniques are only applied for cryptographic implementations (AES, RSA, ECDSA, etc.) and specific cache attacks (F+R, F+F, P+P). In addition, they are focused on process-specific counters to detect the attacks however, the number of processes in a system is high. The performance of proposed techniques in real-world scenarios (noisy environment, multiple concurrent processes) against transient execution attacks (Meltdown, Spectre, Zombieload, etc.) and Rowhammer is questionable. In order to evaluate 4 proposed methods and *FortuneTeller*, we collected 6 million samples (4 million benign executions, 2 million attack executions) with 1ms sampling rate from 10 benign processes and 7 microarchitectural attacks by using **system-wide counters**. Note that each benign and attack execution is monitored 100 times in the server environment. The benign processes are chosen from a diverse set of applications such as Apache, MySQL, browser and cryptographic implementations. The attacks cover cache-based, transient execution and Rowhammer attacks given in Appendix, Table A.7. The detection algorithms from previous works are rewritten in the Matlab environment and tested with the collected data.

CPD from Briongos et al. [30] The first approach is Change Point Detection (CPD) which was implemented by Briongos et al. [30] to detect the anomalies. CPD has the capability of self-learning by observing the number of cache misses. The method has an assumption of no cache miss at the beginning of the learning

process which yields to high number of false positives initially. Even though we increase the initial value of cache misses under attack (μ_a), we still observe several false positives at the beginning. It is also unlikely to monitor each PID in the system since there are hundreds of processes running at the same time.

For the evaluation of CPD method, we use the initial value of $\mu_a = 100$ and $\beta = 0.65$. When the CPD method is applied to our dataset, we observe that the FPR is 3% and FNR is 10% which gives an F-score of 0.9372. However, with the increasing number of concurrent processes, the false positive rate increases in parallel. While this score shows that the CPD method is efficient for low system load, it gives more false positives with the increasing workload. The estimated detection time is around 300 ms for attack executions. The detection performance for Rowhammer and P+P attacks is poor since the number of cache misses is not high compared to benign processes. Therefore, these two attack types increase the FNR overall.

CloudRadar from Zhang et al. [177] CloudRadar [177] benefits from Dynamic Time Warping (DTW) to detect the cryptographic implementations and then, the LLC hit and miss counters are monitored to detect the attacks. In the first step, DTW is used to compare the test data and the signature of cryptographic implementations obtained from branch instructions. Secondly, when the distance between test and target execution is very small, the LLC hit and LLC miss counters are monitored. If there is a sudden jump in these two counters, the anomaly flag is set. Again, this approach requires the PID of the monitored process.

In our dataset, CloudRadar can detect the applications with 100% success rate in a noiseless environment. However, when there are concurrent processes running in the system, DTW distance gets higher since branch instruction counter becomes more noisy. In anomaly detection step, if another concurrent load starts running

at the same time, the cache miss and hit counters start increasing, which results in significant increases of the FPR. Since there is only a simple threshold approach to detect the attacks and the proposed decision window (5 ms) is too small, the FPR rises. In these circumstances, the approach achieves 10% FPR. The attack detection is also not great since it is not possible to detect F+F attack with cache miss and hit counters. Thus, the FNR increases in parallel which yields to 20% FNR. Overall, the detection technique has 0.8572 F-score.

PDF from Chiappetta et al. [35] In the third study, we evaluate the performance of normal distribution and probability density function, which is proposed by Chiappetta et al. [35]. In this technique, five counters (total instructions, CPU cycles, L2 hits, L3 miss and L3 hits) are monitored to detect anomalies in cryptographic implementations. It is an unsupervised approach by learning the normal distribution of the attack execution (F+R) with its mean and variance in the system. After the normal distribution is calculated, the probability density function (pdf) of both attack and benign executions is calculated for each counter sample. Then, an optimal threshold (ϵ) is chosen to separate the benign and attack processes. In our dataset, the results indicated that total instructions and L2 hits decrease the detection rate. The main drawback is that there is not any sophisticated learning process. Therefore, when there is a benign application with high variance and mean, it is more likely to be classified as an anomaly. Especially, Apache server benchmark and videos running in browsers give higher FPR. Moreover, P+P, F+F, and Rowhammer attacks are not detected with a high accuracy, which yields to the F-score of 0.7278 overall.

OC-SVM from Mushtaq et al. [123] The last method to compare is One-Class Support Vector Machine (OC-SVM), which is used by Mushtaq et al. [123] to detect the anomalies on cryptographic implementations. Their scope is limited

to F+F and F+R attacks. The number of counters tested in [123] is higher than three, which makes it impossible to monitor all of them concurrently. Therefore, we chose three counters (L1 miss, L3 hit and L3 total cache access), which give the highest F-score. Even though OC-SVM was used in a supervised way in [123], we used it in an unsupervised manner to maintain the consistency in the comparison. In the training phase, the model is trained with the 50% of the benign execution data. Then, the attack and benign dataset are tested with the trained model. The obtained confidence scores are used to find the optimal decision boundary to separate the benign and attack executions. The optimal decision boundary shows that the FPR and FNR are 0.2750 and 0.2778, respectively. The main problem is that OC-SVM is not sufficient to learn the diverse benign applications, which increases the FPR drastically. Moreover, Rowhammer and F+F attacks are not detected, which is the reason for higher FNR. Therefore, the F-score remains at 0.7240.

FortuneTeller Finally, we apply *FortuneTeller* to detect the anomalies in the system. Since the diversity of the benign executions is smaller in the comparison dataset, it is easier to learn the patterns. It is also important to note that 50 measurements from each benign application is enough to reach the minimum prediction error. Once the LSTM model is trained with the benign applications, the attack executions and remaining benign application data are tested. The FPR and FNR remains at 0.2% and 0.4%, respectively. The F-score is 0.997 for the *FortuneTeller*.

Table 9.2: Comparison of previous methods

	Technique	**F-score**
Briongos et al. [30]	CPD	0.9372
Zhang et al. [177]	DTW	0.8572
Chiappetta et al. [35]	Normal Dist.	0.7278
Mushtaq et al. [123]	OC-SVM	0.7240
Our work	**LSTM/GRU**	**0.9970**

The comparison results are summarized in Table 9.2. The lack of appropriate learning is significant in the wild. It is also obvious that even simple learning algorithm such as CPD can help to outperform other detection techniques. We also show that the detection accuracy increases by learning the sequential patterns of benign applications with the system-wide profiling. Therefore, it is significantly important to extract the fine-grained information from the hardware counters to achieve low FPR and FNR. The common deficiencies of previous works are: (1) only cryptographic implementations are considered, (2) latest attacks such as Rowhammer, Spectre, Meltdown and Zombieload are not considered, (3) no advanced learning technique, (4) the workload is not realistic.

9.4 Discussion

Bypassing FortuneTeller A common way of bypassing *FortuneTeller* is that introducing delays between the attack steps to keep the counter values below the anomaly threshold. To test the robustness of *FortuneTeller*, we inserted different amounts of idle time frames (1μs-1ms) between attack steps in F+F, P+P, and F+R. We observed that the prediction errors of GRU and LSTM networks increase in parallel with the amount of sleep due to the high fluctuation. This shows that introducing delays between attack steps is not an efficient way to circumvent *FortuneTeller*. The reason behind this is the fluctuation in the time series data is not predicted well in the prediction phase. Therefore, we concluded that putting a different amount of sleep between the attack steps is not enough to fool *FortuneTeller*. On the other hand, crafting adversarial examples is an efficient way to bypass Deep Learning-based detection methods. For instance, Rosenberg et al [135] shows that LSTM/GRU based malware detection techniques can be bypassed by

carefully inserting additional API calls in between. Therefore, crafting adversarial code snippets to change the performance counters in the attack code may fool *FortuneTeller*. The main difficulty in this approach is that it is not possible to decrease the counter values by executing more instructions between attack steps. Therefore, applying adversarial examples on hardware counter values is not trivial.

Training Algorithm *FortuneTeller* investigates both available long-term dependency learning techniques. We observed that GRU performs worse than LSTM networks to predict the counter values in the next time steps. This is because of the lack of internal memory state, which keeps the relevant information from previous cells. This result is also supported by the high FPR and FNR of GRU networks. Since the prediction error increases for attack executions more than benign applications, the detection accuracy decreases. Therefore, we recommend to train LSTM networks for microarchitectural attack detection techniques.

Dynamic Detection The current implementation requires to have a GPU to train *FortuneTeller*, as GPU based training 40 times faster than CPU based training. The training is mostly done in an offline phase and it does not affect the dynamic detection. In contrast, dynamic detection heavily depends on the matrix multiplication, since the trained model is loaded as a matrix in the system and the same matrix is multiplied with the current counter values. Hence, the required time to predict the next counter values is low. Besides, we observed that the performance overhead is negligible for the matrix multiplication in the CPU systems. Therefore, *FortuneTeller* can be implemented in server/cloud/laptop environments, even if there is no GPU integrated in the system.

Performance Counter Data Collection Recently, Das et al. [39] showed that performance counters would yield misleading values on various counters. Therefore,

we consider the suggestions proposed in their paper in our study. For instance, we measure the counter values for all the processes in the system so that there is no need for per-process filtering. Additionally, Intel PCM tool [84] uses interrupt-based counters hence, the context switches and page faults have no effect on our experiments for system-wide profiling.

9.5 Outcome

This study presented *FortuneTeller*, which exploits the power of neural networks to overcome the limitations of the prior works, and further proposes a novel generic model to classify microarchitectural events. *FortuneTeller* is able to dynamically detect microarchitectural anomalies in the system through learning benign workload. In our study, we adopted two state-of-the-art RNN models: GRU and LSTM. We concluded that LSTM is more preferable compared to GRU for our use case. Further, the number of measurements has significant effect on the validation error in the training phase, which makes it crucial to choose the optimal values to have better prediction results. *FortuneTeller* is applicable to both server and laptop environments with a high accuracy. To evaluate the performance of *FortuneTeller*, we used both benchmarks and real-world applications and achieved 0.05% and 0.02% FPRs in one minute for server and laptop environments, respectively. *FortuneTeller* is also tested against previous works in the realistic scenarios and it is concluded that *FortuneTeller* outperforms other detection mechanisms in the wild.

Chapter 10

Conclusion

Microarchitectural attacks threaten the security and privacy of cloud, computer, and mobile phone users. This threat becomes more dangerous when automated information extraction techniques such as Machine Learning are applied to side-channel traces.

In this dissertation, we show that cross-VM leakage exists in public clouds and can be exploited as an attack vector to steal cryptographic keys. Furthermore, the privacy of cloud service clients can be violated from co-located VMs by implementing cache-based attacks. These results show that necessary steps should be taken by chip vendors to secure the entire cache architecture in multi-core systems.

We also demonstrated that performance counter profiling attacks are applicable in modern personal computers. Any malicious application in the system can monitor counters to detect the visited websites in privacy protected browsers. Moreover, the detection rate is improved by analyzing the measurements with Machine Learning techniques.

The privacy violations on mobile phones are also investigated by applying cache attacks. While we showed that applications, websites, and videos can be identified

by collecting cache traces in mobile phones, Deep Learning techniques achieve higher detection rates compared to traditional Machine Learning algorithms.

Finally, we introduce a Recurrent Neural Network based method to detect the ongoing microarchitectural attacks in the system. This work is the first one to show that single Machine Learning model can be trained to detect multiple attacks which is also more prune to system noise. It has been shown that our technique outperforms other detection tools in a number of real-life applications.

Bibliography

[1] Amazon EC2 Instances. http://aws.amazon.com/ec2/instance-types/.

[2] AWS IP Address Ranges. https://ip-ranges.amazonaws.com/ip-ranges.json.

[3] Fix Flush and Reload in RSA. https://lists.gnupg.org/pipermail/gnupg-announce/2013q3/000329.html.

[4] Global Cloud Index Projects Cloud Traffic to Represent 95 Percent of Total Data Center Traffic by 2021. https://newsroom.cisco.com/press-release-content?type=webcontent&articleId=1908858. Accessed: 2017-5-25.

[5] Google Compute Engine Instance Types. https://cloud.google.com/compute/docs/machine-types.

[6] Intel Xeon 2670-v2. http://ark.intel.com/es/products/75275/Intel-Xeon-Processor-E5-2670-v2-25M-Cache-2_50-GHz.

[7] Leak site directory. http://www.leakdirectory.org/index.php/Leak_Site_Directory.

[8] More buying, less building in The Age of Consumption. https://451research.com/blog/1933-more-buying-less-building-in-the-age-of-consumption. Accessed: 2017-5-24.

[9] OpenSSL fix flush and reload ECDSA nonces. https://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=2198be3483259de374f91e57d247d0fc667aef29.

[10] Phoronix Test Suite Tests. https://openbenchmarking.org/tests/pts.

[11] Transparent Page Sharing: additional management capabilities and new default settings. http://blogs.vmware.com/security/vmware-security-response-center/page/2.

[12] Worldwide Public Cloud Services Spending Forecast to Reach $210 Billion This Year, According to IDC. https://www.idc.com/getdoc.jsp?containerId=prUS44891519. Accessed: 2017-5-25.

[13] Kernel Samepage Merging. http://kernelnewbies.org/Linux_2_6_32#head-d3f32e41df508090810388a57efce73f52660ccb/, April 2014.

[14] Study Says Self-Driving Cars Are Safer Than Human-Driven Vehicles: Should You Believe It? http://bit.ly/1mTiDWk, Jan 2016. Tech Times.

[15] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 265–283.

[16] Aciiçmez, O. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (2007), CSAW '07, ACM, pp. 11–18.

[17] Aciiçmez, O., Brumley, B. B., and Grabher, P. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems* (Berlin, Heidelberg, 2010), CHES'10, Springer, pp. 110–124.

[18] Aldaya, A. C., Brumley, B. B., ul Hassan, S., García, C. P., and Tuveri, N. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 870–887.

[19] Alexa Internet Inc. The top 500 sites on the web, 2018. http://www.alexa.com/topsites. Last accessed 2018-01-01.

[20] Alphago. https://deepmind.com/research/alphago/.

[21] Android Open Source Project. Bionic Initial Contribution, 2008. https://android.googlesource.com/platform/bionic/+/a27d2baa. Last accessed 2019-01-21.

[22] Bahador, M. B., Abadi, M., and Tajoddin, A. Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on* (2014), IEEE, pp. 703–708.

[23] Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., and Butler, K. Detecting co-residency with active traffic analysis techniques. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop* (2012), pp. 1–12.

[24] Beltramelli, T., and Risi, S. Deep-spying: Spying using smartwatch and deep learning. *CoRR abs/1512.05616* (2015).

[25] Benger, N., van de Pol, J., Smart, N., and Yarom, Y. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems – CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, L. Batina and M. Robshaw, Eds., vol. 8731 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 75–92.

[26] Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems – CHES 2014* (Berlin, Heidelberg, 2014), Springer, pp. 75–92.

[27] Bernstein, D. J., Chang, Y.-A., Cheng, C.-M., Chou, L.-P., Heninger, N., Lange, T., and Van Someren, N. Factoring rsa keys from certified smart cards: Coppersmith in the wild. In *Advances in Cryptology-ASIACRYPT 2013*. Springer, 2013, pp. 341–360.

[28] Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., and Kurmus, A. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 785–800.

[29] Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)* (2016), IEEE, pp. 987–1004.

[30] Briongos, S., Irazoqui, G., Malagón, P., and Eisenbarth, T. Cacheshield: Detecting cache attacks through self-observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2018), CODASPY '18, ACM, pp. 224–235.

[31] Brumley, B. B., and Hakala, R. M. Cache-timing template attacks. In *Advances in Cryptology – ASIACRYPT 2009* (Berlin, Heidelberg, 2009), M. Matsui, Ed., Springer Berlin Heidelberg, pp. 667–684.

[32] Cauligi, S., Soeller, G., Brown, F., Johannesmeyer, B., Huang, Y., Jhala, R., and Stefan, D. Fact: A flexible, constant-time programming language. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017* (2017), pp. 69–76.

[33] Chandrashekar, G., and Sahin, F. A survey on feature selection methods. *Computers & Electrical Engineering 40*, 1 (2014), 16–28.

[34] Chang, C.-C., and Lin, C.-J. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST) 2*, 3 (2011), 27.

[35] Chiappetta, M., Savas, E., and Yilmaz, C. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing 49* (2016), 1162–1174.

[36] Chiu, C.-C., Sainath, T. N., Wu, Y., Prabhavalkar, R., Nguyen, P., Chen, Z., Kannan, A., Weiss, R. J., Rao, K., Gonina, E., et al. State-of-the-art speech recognition with sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2018), IEEE, pp. 4774–4778.

[37] Chollet, F., et al. Keras, 2015.

[38] Craig Labovitz. How Big is Amazons Cloud? `http://www.deepfield.com/2012/04/how-big-is-amazons-cloud/`, 2012.

[39] Das, S., Werner, J., Antonakakis, M., Polychronakis, M., and Monrose, F. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security* (2019), IEEE, p. 0.

[40] Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 559–570.

[41] Dhem, J.-F., Koeune, F., Leroux, P.-A., Mestré, P., Quisquater, J.-J., and Willems, J.-L. A Practical Implementation of the Timing Attack. In *Smart Card Research and Applications*, J.-J. Quisquater and B. Schneier, Eds., vol. 1820 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, pp. 167–182.

[42] Diao, W., Liu, X., Li, Z., and Zhang, K. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 414–432.

[43] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Piscataway, NJ, USA, 2016), MICRO-49, IEEE Press, pp. 40:1–40:13.

[44] Evtyushkin, D., Riley, R., Abu-Ghazaleh, N. C., ECE, and Ponomarev, D. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 693–707.

[45] Feizollah, A., Anuar, N. B., Salleh, R., Amalina, F., Shamshirband, S., et al. A study of machine learning classifiers for anomaly-based mobile botnet detection. *Malaysian Journal of Computer Science 26*, 4 (2013), 251–265.

[46] Felten, E. W., and Schneider, M. A. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2000), CCS '00, ACM, pp. 25–32.

[47] Forbes. 83% Of Enterprise Workloads Will Be In The Cloud By 2020.

[48] Frigo, P., Vannacci, E., Hassan, H., van der Veen, V., Mutlu, O., Giuffrida, C., Bos, H., and Razavi, K. Trrespass: Exploiting the many sides of target row refresh. *arXiv preprint arXiv:2004.01807* (2020).

[49] Gaudin, S. Public cloud market ready for 'hypergrowth' period. Computerworld Article, April 2014. `http://www.computerworld.com/article/2488572/cloud-computing/public-cloud-market-ready-for--hypergrowth--period.html`.

[50] Genkin, D., Pachmanov, L., Pipman, I., and Tromer, E. Stealing keys from pcs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *International Workshop on Cryptographic Hardware and Embedded Systems* (Berlin, Heidelberg, 2015), Springer, Springer, pp. 207–228.

[51] Genkin, D., Pachmanov, L., Tromer, E., and Yarom, Y. Drive-by key-extraction cache attacks from portable code. Cryptology ePrint Archive, Report 2018/119, 2018. `https://eprint.iacr.org/2018/119`.

[52] Genkin, D., Shamir, A., and Tromer, E. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference* (Berlin, Heidelberg, 2014), Springer, Springer, pp. 444–461.

[53] Ghose, T. All in: Artificial intelligence beats the world's best poker players. `https://www.livescience.com/57717-artificial-intelligence-wins-texas-hold-em.html`, Feb 2017.

[54] Godfrey, M. M., and Zulkernine, M. A server-side solution to cache-based the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013* (2013), pp. 163–170.

[55] Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. 2016.

[56] Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[57] Gras, B., Razavi, K., Bos, H., and Giuffrida, C. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 955–972.

[58] Gruss, D., Bidner, D., and Mangard, S. Practical memory deduplication attacks in sandboxed javascript. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 108–122.

[59] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems* (2017), Springer, pp. 161–176.

[60] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 368–379.

[61] Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (2016), Springer, pp. 300–321.

[62] Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A remote software-induced fault attack in javascript. In *13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2016).

[63] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug 2015), USENIX Association, pp. 897–912.

[64] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security*

Symposium (USENIX Security 15) (Washington, D.C., 2015), USENIX Association, pp. 897–912.

[65] Gullasch, D., Bangerter, E., and Krenn, S. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Oakland, CA, USA, 2011), SP '11, IEEE Computer Society, pp. 490–505.

[66] Gulmezoglu, B., Eisenbarth, T., and Sunar, B. Cache-based application detection in the cloud using machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (New York, NY, USA, 2017), ASIA CCS '17, ACM, pp. 288–300.

[67] Gülmezoğlu, B., Inci, M. S., Irazoqui, G., Eisenbarth, T., and Sunar, B. A faster and more realistic flush+ reload attack on aes. In *International Workshop on Constructive Side-Channel Analysis and Secure Design* (2015), Springer, pp. 111–126.

[68] Gulmezoglu, B., Inci, M. S., Irazoqui, G., Eisenbarth, T., and Sunar, B. Cross-vm cache attacks on aes. *IEEE Transactions on Multi-Scale Computing Systems 2*, 3 (2016), 211–222.

[69] Gulmezoglu, B., Moghimi, A., Eisenbarth, T., and Sunar, B. Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning. *arXiv preprint arXiv:1907.03651* (2019).

[70] Gülmezoglu, B., Zankl, A., Eisenbarth, T., and Sunar, B. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II* (2017), pp. 80–97.

[71] Gulmezoglu, B., Zankl, A., Tol, C., Islam, S., Eisenbarth, T., and Sunar, B. Undermining user privacy on mobile devices using ai. *arXiv preprint arXiv:1811.11218* (2018).

[72] Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 205–220.

[73] Herath, N., and Fogh, A. These are not your grand daddys cpu performance counters–cpu hardware performance counters for security. *Black Hat Briefings* (2015).

[74] Horn, J. speculative execution, variant 4: speculative store bypass, 2018.

[75] HORNBY, T. Side-channel attacks on everyday applications: Distinguishing inputs with flush+reload. Black Hat USA, 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Hornby-Side-Channel-Attacks-On-Everyday-Applications-wp.pdf.

[76] HOSPODAR, G., GIERLICHS, B., DE MULDER, E., VERBAUWHEDE, I., AND VANDEWALLE, J. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering 1*, 4 (2011), 293–302.

[77] HU, W. Reducing timing channels with fuzzy time. In *IEEE Symposium on Security and Privacy* (1991), pp. 8–20.

[78] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 191–205.

[79] IGNATOV, A., TIMOFTE, R., CHOU, W., WANG, K., WU, M., HARTLEY, T., AND GOOL, L. V. AI benchmark: Running deep neural networks on android smartphones. *CoRR abs/1810.01109* (2018).

[80] INCI, M. S., EISENBARTH, T., AND SUNAR, B. Deepcloak: Adversarial crafting as a defensive measure to cloak processes. *arXiv preprint arXiv:1808.01352* (2018).

[81] INCI, M. S., GÜLMEZOGLU, B., APECECHEA, G. I., EISENBARTH, T., AND SUNAR, B. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. *IACR Cryptology ePrint Archive 2015*, 1-15 (2015).

[82] INCI, M. S., GULMEZOGLU, B., EISENBARTH, T., AND SUNAR, B. Co-location detection on the cloud. In *International Workshop on Constructive Side-Channel Analysis and Secure Design* (2016), Springer, pp. 19–34.

[83] İNCI, M. S., GULMEZOGLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cache attacks enable bulk key recovery on the cloud. In *Cryptographic Hardware and Embedded Systems* (Berlin, Heidelberg, 2016), pp. 368–388.

[84] INTEL. Intel pcm, 2019. https://github.com/opcm/pcm. Last accessed 2019-02-15.

[85] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2015), SP '15, IEEE Computer Society, pp. 591–604.

[86] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$a: A shared cache attack that works across cores and defies vm sandboxing and its application to aes. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (San Jose, CA, USA, 2015), SP '15, IEEE Computer Society, pp. 591–604.

[87] Irazoqui, G., Eisenbarth, T., and Sunar, B. Mascat: Stopping microarchitectural attacks before execution. *IACR Cryptology ePrint Archive 2016* (2016), 1196.

[88] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Fine grain cross-vm attacks on xen and vmware are possible! Cryptology ePrint Archive, Report 2014/248, 2014. http://eprint.iacr.org/.

[89] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a minute! a fast, cross-vm attack on aes. In *Research in Attacks, Intrusions and Defenses* (2014), Springer, pp. 299–319.

[90] Irazoqui, G., IncI, M. S., Eisenbarth, T., and Sunar, B. Know thy neighbor: Crypto library detection in cloud. *Proceedings on Privacy Enhancing Technologies 2015*, 1 (2015), 25–40.

[91] Islam, S., Moghimi, A., Bruhns, I., Krebbel, M., Gulmezoglu, B., Eisenbarth, T., and Sunar, B. {SPOILER}: Speculative load hazards boost rowhammer and cache attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), pp. 621–637.

[92] Jackson, C., Bortz, A., Boneh, D., and Mitchell, J. C. Protecting browser state from web privacy attacks. In *Proceedings of the 15th International Conference on World Wide Web* (New York, NY, USA, 2006), WWW '06, ACM, pp. 737–744.

[93] Jaleel, A., Theobald, K. B., Jr., S. C. S., and Emer, J. S. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of ISCA 2010, June 19-23, 2010, Saint-Malo, France* (2010), pp. 60–71.

[94] Jana, S., and Shmatikov, V. Memento: Learning secrets from process footprints. In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 143–157.

[95] Jones, M. T. Anatomy of linux kernel shared memory. http://www.ibm.com/developerworks/linux/library/l-kernel-shared-memory/l-kernel-shared-memory-pdf.pdf/, April 2010.

[96] Kim, H., Lee, S., and Kim, J. Inferring browser activity and status through remote monitoring of storage usage. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications* (New York, NY, USA, 2016), ACSAC '16, ACM, pp. 410–421.

[97] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints* (Jan. 2018).

[98] Kocher, P., Jaffe, J., and Jun, B. Differential power analysis. In *Advances in cryptologyCRYPTO99* (1999), Springer, pp. 789–789.

[99] Kocher, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology — CRYPTO '96* (Berlin, Heidelberg, 1996), Springer, pp. 104–113.

[100] Koruyeh, E. M., Khasawneh, K. N., Song, C., and Abu-Ghazaleh, N. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)* (Baltimore, MD, Aug. 2018), USENIX Association.

[101] Lee, S., Kim, Y., Kim, J., and Kim, J. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *2014 IEEE Symposium on Security and Privacy (SP)* (2014), IEEE, pp. 19–33.

[102] Lerman, L., Bontempi, G., and Markowitch, O. Side channel attack: an approach based on machine learning. *Center for Advanced Security Research Darmstadt* (2011), 29–41.

[103] Liang, B., You, W., Liu, L., Shi, W., and Heiderich, M. Scriptless timing attacks on web browser privacy. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2014), pp. 112–123.

[104] Libgcrypt. The Libgcrypt reference manual. `http://www.gnupg.org/documentation/manuals/gcrypt/`.

[105] Linux Kernel Developers. perf: Linux profiling with performance counters, 2015. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[106] Linux Programmer's Manual. perf_event_open - set up performance monitoring. `http://man7.org/linux/man-pages/man2/perf_event_open.2.html`, 2016. Accessed: 2017-06-29.

[107] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), pp. 549–564.

[108] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug 2016), USENIX Association, pp. 549–564.

[109] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium* (Baltimore, MD, 2018).

[110] Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., and Lee, R. B. 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.

[111] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (San Jose, CA, USA, 2015), SP '15, IEEE Computer Society, pp. 605–622.

[112] LLVM. Speculative load hardening, 2019. `https://llvm.org/docs/SpeculativeLoadHardening.html`. Last accessed 2020-04-07.

[113] Maghrebi, H., Portigliatti, T., and Prouff, E. Breaking cryptographic implementations using deep learning techniques. In *Security, Privacy, and Applied Cryptography Engineering* (Cham, 2016), C. Carlet, M. A. Hasan, and V. Saraswat, Eds., Springer International Publishing, pp. 3–26.

[114] Maisuradze, G., and Rossow, C. ret2spec. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Jan 2018).

[115] Malone, C., Zahran, M., and Karri, R. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the sixth ACM workshop on Scalable trusted computing* (2011), ACM, pp. 71–76.

[116] Mambretti, A., Sandulescu, A., Sorniotti, A., Robertson, W., Kirda, E., and Kurmus, A. Bypassing memory safety mechanisms through speculative control flow hijacks. *arXiv preprint arXiv:2003.05503* (2020).

[117] Mangard, S., Oswald, E., and Popp, T. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security).* Springer, Berlin, Heidelberg, 2007.

[118] Martinasek, Z., Hajny, J., and Malina, L. Optimization of power analysis using neural network. In *Smart Card Research and Advanced Applications*

(Cham, 2014), A. Francillon and P. Rohatgi, Eds., Springer International Publishing, pp. 94–107.

[119] Martinasek, Z., and Zeman, V. Innovative method of the power analysis. *Radioengineering 22*, 2 (2013), 586–594.

[120] Microsoft. Performance counters (windows), 2017. https://msdn.microsoft.com/de-de/library/windows/desktop/aa373083(v=vs.85).aspx.

[121] Minkin, M., Moghimi, D., Lipp, M., Schwarz, M., Van Bulck, J., Genkin, D., Gruss, D., Piessens, F., Sunar, B., and Yarom, Y. Fallout: Reading kernel writes from user space. *arXiv preprint arXiv:1905.12701* (2019).

[122] Moghimi, A., Eisenbarth, T., and Sunar, B. Memjam: A false dependency attack against constant-time crypto implementations in SGX. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings* (2018), pp. 21–44.

[123] Mushtaq, M., Akram, A., Bhatti, M. K., Chaudhry, M., Lapotre, V., and Gogniat, G. Nights-watch: a cache-based side-channel intrusion detector using hardware performance counters. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy* (2018), ACM, p. 1.

[124] Naghibijouybari, H., Neupane, A., Qian, Z., and Abu-Ghazaleh, N. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 2139–2153.

[125] Nilsson, N. J. Learning machines.

[126] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA), CCS '15, pp. 1406–1418.

[127] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (NY, USA, 2015), CCS, ACM, pp. 1406–1418.

[128] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: The case of aes. In *Topics in Cryptology – CT-RSA 2006* (Berlin, Heidelberg, 2006), Springer, pp. 1–20.

[129] Pan, Y., Shen, P., and Shen, L. Speech emotion recognition using support vector machine. *International Journal of Smart Home 6*, 2 (2012), 101–108.

[130] Panchenko, A., Lanze, F., Pennekamp, J., Engel, T., Zinnen, A., Henze, M., and Wehrle, K. Website fingerprinting at internet scale. In *NDSS* (2016).

[131] Peng, L. Detecting diabetic eye disease with machine learning. `https://blog.google/topics/machine-learning/detecting-diabetic-eye-disease-machine-learning/`, Nov 2016.

[132] Percival, C. Cache missing for fun and profit, 2005.

[133] Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., and Bos, H. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security symposium* (2016), pp. 1–18.

[134] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.

[135] Rosenberg, I., Shabtai, A., Rokach, L., and Elovici, Y. Generic black-box end-to-end attack against rnns and other api calls based malware classifiers. *arXiv preprint arXiv:1707.05970* (2017).

[136] Sak, H., Senior, A., and Beaufays, F. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128* (2014).

[137] Schuster, R., Shmatikov, V., and Tromer, E. Beauty and the burst: Remote identification of encrypted video streams. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 1357–1374.

[138] Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., and Gruss, D. Zombieload: Cross-privilege-boundary data sampling. *arXiv preprint arXiv:1905.05726* (2019).

[139] Schwarz, M., Maurice, C., Gruss, D., and Mangard, S. Fantastic timers and where to find them: High-resolution microarchitectural attacks

in javascript. In *Financial Cryptography and Data Security* (Cham, 2017), A. Kiayias, Ed., Springer International Publishing, pp. 247–267.

[140] Schwarz, M., Schwarzl, M., Lipp, M., Masters, J., and Gruss, D. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security* (2019), Springer, pp. 279–299.

[141] Seaborn, M., and Dullien, T. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat 15* (2015).

[142] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems 38*, 1 (2012), 161–190.

[143] Shusterman, A., Kang, L., Haskal, Y., Meltser, Y., Mittal, P., Oren, Y., and Yarom, Y. Robust website fingerprinting through the cache occupancy channel. *CoRR abs/1811.07153* (2018).

[144] Spreitzer, R., Kirchengast, F., Gruss, D., and Mangard, S. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (2018), ACM, pp. 749–763.

[145] Spreitzer, R., Palfinger, G., and Mangard, S. Scandroid: Automated side-channel analysis of android apis. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2018), ACM, pp. 224–235.

[146] Steven Levy. The brain is here and it is already inside your phone, 2014. `https://www.wired.com/2016/08/an-exclusive-look-at-how-ai-and-machine-learning-work-at-apple/`. Last accessed 2019-02-02.

[147] Sundermeyer, M., Schlüter, R., and Ney, H. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association* (2012).

[148] Suzaki, K., Iijima, K., Toshiki, Y., and Artho, C. Implementation of a memory disclosure attack on memory deduplication of virtual machines. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences 96*, 1 (2013), 215–224.

[149] Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Memory deduplication as a threat to the guest OS. In *Proceedings of the Fourth European Workshop on System Security* (2011), ACM, p. 1.

[150] Tang, A., Sethumadhavan, S., and Stolfo, S. Unsupervised anomaly-based malware detection using hardware features. In *Research in Attacks, Intrusions and Defenses*. 2014, pp. 109–129.

[151] The Verge. Google announces over 2 billion monthly active devices on android, 2018. `https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users`. Last accessed 2017-02-01.

[152] Tromer, E., Osvik, D. A., and Shamir, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* (2010).

[153] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium* (2018), pp. 991–1008.

[154] Van Bulck, J., Moghimi, D., Schwarz, M., Lipp, M., Minkin, M., Genkin, D., Yarom, Y., Sunar, B., Gruss, D., and Piessens, F. Lvi: Hijacking transient execution through microarchitectural load value injection. In *41th IEEE Symposium on Security and Privacy (S&P20)* (2020), pp. 1399–1417.

[155] van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1675–1689.

[156] van Schaik, S., Milburn, A., sterlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. RIDL: Rogue in-flight data load. In *S&P* (May 2019).

[157] Varadarajan, V., Ristenpart, T., and Swift, M. Scheduler-based defenses against cross-vm side-channels. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014* (San Diego, CA, 2014), K. Fu and J. Jung, Eds., USENIX Association, pp. 687–702.

[158] Varadarajan, V., Zhang, Y., Ristenpart, T., and Swift, M. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C.), USENIX Association, pp. 913–928.

[159] Vila, P., and Kopf, B. Loophole: Timing attacks on shared event loops in chrome. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 849–864.

[160] Vila, P., Köpf, B., and Morales, J. F. Theory and practice of finding eviction sets. *CoRR abs/1810.01497* (2018).

[161] VMWare. Understanding Memory Resource Management in VMware vSphere 5.0. `http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf`.

[162] Waldspurger, C. A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

[163] Wang, X., and Karri, R. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE* (2013), IEEE, pp. 1–7.

[164] Wang, X., Konstantinou, C., Maniatakos, M., and Karri, R. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (2015), IEEE Press, pp. 544–551.

[165] Weinberg, Z., Chen, E. Y., Jayaraman, P. R., and Jackson, C. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *2011 IEEE Symposium on Security and Privacy* (May 2011), pp. 147–161.

[166] Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., and Sigl, G. Data–differential address trace analysis: finding address-based side-channels in binaries. In *27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD* (2018), pp. 603–620.

[167] Wichelmann, J., Moghimi, A., Eisenbarth, T., and Sunar, B. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference* (2018), ACM, pp. 161–173.

[168] Xia, Y., Liu, Y., Chen, H., and Zang, B. Cfimon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on* (2012), IEEE, pp. 1–12.

[169] Xiao, Y., Zhang, X., Zhang, Y., and Teodorescu, R. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium* (2016), pp. 19–35.

[170] Xu, Z., Wang, H., and Wu, Z. A measurement study on co-residence threat inside the cloud. In *24th USENIX Security* (2015), pp. 929–944.

[171] Yarom, Y., and Benger, N. Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. *IACR Cryptology ePrint Archive 2014* (2014), 140.

[172] Yarom, Y., and Falkner, K. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), pp. 719–732.

[173] Yarom, Y., and Falkner, K. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 719–732.

[174] Yarom, Y., Genkin, D., and Heninger, N. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering 7*, 2 (2017), 99–112.

[175] Yuan, L., Xing, W., Chen, H., and Zang, B. Security breaches as pmu deviation: detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (2011), ACM, p. 6.

[176] Zankl, A., Seuschek, H., Irazoqui, G., and Gulmezoglu, B. side-channel attacks in the internet of things: threats and challenges. In *Solutions for Cyber-Physical Systems Ubiquity*. IGI Global, 2018, pp. 325–357.

[177] Zhang, T., Zhang, Y., and Lee, R. B. Cloudradar: A real-time side-channel attack detection system in clouds. F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds., pp. 118–140.

[178] Zhang, X., Wang, X., Bai, X., Zhang, Y., and Wang, X. Os-level side channels without procfs: Exploring cross-app information leakage on ios. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS 2018). Internet Society* (2018).

[179] Zhang, X., Xiao, Y., and Zhang, Y. 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* (New York, NY, USA, 2016), CCS '16, ACM, pp. 858–870.

[180] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE Symposium on Security and Privacy* (May 2011), pp. 313–328.

[181] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), ACM, ACM, pp. 305–316.

[182] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), pp. 305–316.

[183] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 990–1003.

[184] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 990–1003.

[185] Zheng, Y., Liu, Q., Chen, E., Ge, Y., and Zhao, J. L. Time series classification using multi-channels deep convolutional neural networks. In *Web-Age Information Management* (Cham, 2014), F. Li, G. Li, S.-w. Hwang, B. Yao, and Z. Zhang, Eds., Springer International Publishing, pp. 298–310.

Appendix A

A.1 List of the Websites

Table A.1: List of websites profiled PerfWeb.

Website Number in the Figures	
1) Netflix.com	21) Office.com
2) Amazon.com	22) Microsoftonline.com
3) Facebook.com	23) Chase.com
4) Google.com	24) Nytimes.com
5) Yahoo.com	25) Blogspot.com
6) Youtube.com	26) Paypal.com
7) Wikipedia.org	27) Imdb.com
8) Reddit.com	28) Wordpress.com
9) Twitter.com	29) Espn.com
10) Ebay.com	30) Wikia.com
11) Linkedin.com	31) Wikileaks.org
12) Diply.com	32) Aljazeera.com/investigations
13) Instagram.com	33) Balkanleaks.eu
14) Live.com	34) Unileaks.org
15) Bing.com	35) Globaleaks.com
16) Imgur.com	36) Liveleak.com
17) Ntd.tv	37) Globalwitness.org
18) Cnn.com	38) Wikispooks.com
19) Pinterest.com	39) Officeleaks.com
20) Tumblr.com	40) Publeaks.nl

A.2 Additional Tables and Figures for Mobile Phones Attack

This section provides complementary information regarding our experiments. It lists the profiled applications, websites, and videos, and gives the parameters that were explored while constructing our convolutional neural network.

A.2.1 Profiled Applications, Websites, Videos

Table A.4 lists the videos that are profiled in our experiments. YouTube videos are chosen from the list of most watched videos on YouTube. Trailers and recaps viewed in the Netflix app are from the series *The House of Cards*. Tables A.2 and A.3 list the profiled applications and websites. The selection of websites is taken from the Alexa ranking [19].

Table A.2: List of profiled applications.

Applications		
1) Spotify	35) Reddit	69) OurTime
2) Snapchat	36) Imdb	70) HowAboutWe
3) Instagram	37) Creditkarma	71) Tiktok
4) Facebook	38) Alexa	72) Canva
5) YouTube	39) Yahoo	73) Autolist
6) Chrome	40) Starz	74) Sephora
7) Netflix	41) Zedge	75) Indeed
8) Uber	42) Textnow	76) Marvel
9) Twitter	43) Soundcloud	77) Hinge
10) Bitmoji	44) Booking	78) Daylio
11) Google Drive	45) Duolingo	79) Roku
12) Pandora	46) Tinder	80) Investing
13) NY Times	47) Joom	81) Ifood
14) Pinterest	48) Xbox	82) Fitbit
15) Lyft	49) Shazam	83) Goodrx
16) InBrowser	50) Chase	84) Fastnews
17) Firefox Focus	51) Huffington	85) Touchnote
18) Orfox	52) Breitbart	86) Nike
19) Musical Focus	53) Earspy	87) Sony
20) Wish	54) Ispy	88) Kayak
21) Hulu	55) Spycamera	89) Expedia
22) Workout	56) Mspy	90) Sketch
23) Waze	57) Secretagent	91) PlutoTV
24) Walmart	58) Politico	92) Grubhub
25) Wholefoods	59) TheHill	93) McDonald's
26) Dairy Queen	60) Dailykos	94) Target
27) Discord	61) Infowars	95) Trivia
28) Venmo	62) Match	96) Starbucks
29) Groupon	63) PlentyofFish	97) Horoscope
30) Twitch	64) Zoosk	98) Beetles
31) Yelp	65) eHarmony	99) Glasdoor
32) Letgo	66) Okcupid	100)Tickermaster
33) Iheart	67) Badoo	
34) eBay	68) Christian Mingle	

Table A.3: List of profiled websites.

Websites					
1) Google	18) Craigslist	35) Hulu	52) Breitbart	69) BlackPeopleMeet	86) Nginx
2) Facebook	19) Paypal	36) Quora	53) Drudgereport	70) HowAboutWe	87) Springer
3) Wikipedia	20) Apple	37) Salesforce	54) Politico	71) Oracle	88) Apache
4) Amazon	21) Bing	38) Wells	55) The Hill	72) Reuters	89) Flickr
5) Reddit	22) Chase	39) Bank of America	56) Slate	73) BBC	90) Grawatar
6) Yahoo	23) Zillow	40) Stackoverflow	57) Dailykos	74) Nasa	91) Sourceforge
7) Twitter	24) Walmart	41) Guardian	58) Infowars	75) Eventbrite	92) Archive
8) eBay	25) Yelp	42) Forbes	59) Salon	76) Dailymotion	93) Go
9) Netflix	26) Github	43) Dropbox	60) TheBlaze	77) Blogger	94) Wix
10) Linkedin	27) NY Times	44) Mozilla	61) Match	78) Nature	95) Myspace
11) Office	28) Pinterest	45) Soundcloud	62) Plenty of Fish	79) Digg	96) Mysql
12) Cnn	29) Imdb	46) Weebly	63) Zoosk	80) Wiley	97) Time
13) Espn	30) Microsoft	47) Vimeo	64) Okcupid	81) Wired	98) Cnbc
14) Wikia	31) Msn	48) Adobe	65) eHarmony	82) Ted	99) Skype
15) Twitch	32) Fox News	49) Wordpress	66) Badoo	83) Feedburner	100) Alibaba
16) Live	33) Blogspot	50) Tumblr	67) Christian Mingle	84) Oath	
17) Instagram	34) Dailymail	51) Huffington	68) OurTime	85) Ietf	

Table A.4: List of profiled videos.

Youtube (left) and Netflix (right) Videos	
1) Despacito	1) Season 1 Trailer
2) See You Again	2) Season 2 Trailer
3) Shape of You	3) Season 3 Trailer
4) Gangnam Style	4) Season 4 Trailer
5) Uptown Funk	5) Season 5 Trailer
6) Sorry	6) Season 1 Recap
7) Sugar	7) Season 2 Recap
8) Shake it Off	8) Season 3 Recap
9) Roar	9) Season 4 Recap
10) Bailando	10) Season 1 Trailer (Extended)

A.2.2 CNN Parameter Selection

Table A.5 documents the CNN parameter exploration that was conducted prior to our experiments. The final parameters are highlighted in bold.

A.3 Appendix for FortuneTeller

A.3.1 Tables for Performance Counters and Benchmarks

Table A.5: CNN parameter exploration. Final selection highlighted in bold.

Convolution	Max Pooling	Dropout	Kernel Size	Dense	Training Loss	Training Accuracy (%)	Validation Loss	Validation Accuracy (%)
1024	2	0.1	3	50	0.2568	93.54	0.7092	83.04
512	2	0.1	3	50	0.2085	94.23	0.6876	84.03
256	2	0.1	3	50	0.2554	95.01	0.7127	82.75
128	2	0.1	3	50	0.2666	93.56	0.7307	82.78
64	2	0.1	3	50	0.2790	92.31	0.7342	82.68
32	2	0.1	3	50	0.5443	83.23	0.8038	80.56
16	2	0.1	3	50	0.3821	89.04	0.6910	82.38
8	2	0.1	3	50	0.4513	86.94	0.7057	82.29
512-256	2	0.1	3	50	0.2581	92.17	0.6436	84.40
512-128	2	0.1	3	50	0.3725	89.09	0.6510	84.18
512-64	2	0.1	3	50	0.6743	81.12	0.7638	80.93
512-32	2	0.1	3	50	0.4495	87.25	0.7355	81.46
512-256-128	2	0.1	3	50	0.2564	91.24	0.6440	84.55
512-256-64	2	0.1	3	50	0.3345	90.15	0.6609	84.09
512-256-32	2	0.1	3	50	0.3259	90.75	0.6984	81.25
512-256	2	0.1	3	50	0.2581	92.17	0.6436	84.40
512-256	4	0.1	3	50	0.4823	86.48	0.7467	82.45
512-256	8	0.1	3	50	0.5642	84.24	0.7160	81.54
512-256	2	0.2	3	50	0.2581	92.17	0.6436	84.80
512-256	2	0.3	3	50	0.2756	91.24	0.6783	83.34
512-256	2	0.4	3	50	0.2894	90.57	0.6928	82.86
512-256	2	0.5	3	50	0.3184	88.37	0.7293	81.43
512-256	2	0.2	6	50	0.4068	87.65	0.6583	83.45
512-256	2	0.2	9	50	0.3686	89.48	0.6314	85.21
512-256	2	0.2	18	50	0.3079	91.00	0.6794	84.82
512-256	2	0.2	27	50	0.3283	90.06	0.6915	83.87
512-256	2	0.2	9	100	0.3387	90.03	0.6836	83.07
512-256	2	0.2	9	150	0.3208	90.39	0.6456	83.57
512-256	**2**	**0.2**	**9**	**200**	**0.3104**	**91.25**	**0.6218**	**85.76**
512-256	2	0.2	9	250	0.3487	89.74	0.6424	83.38
512-256	2	0.2	9	300	0.3562	88.96	0.6592	82.51
512-256	2	0.2	9	350	0.3859	86.52	0.6834	82.15

Table A.6: Counter Selection for core counters

Counter	F-score
LLC_Miss	**0.8522**
ICACHE.Hit	**0.8154**
ICACHE.Miss	**0.8023**
L1D.Replacement	0.7523
L1D_Pend_Miss.Pending	0.6818
L1D_Pend_Miss.Request_FB_Full	0.6698
L2_Rqsts_Lat_Cache.Miss	0.6244
LLC_Reference	0.6167
Dtlb_Load_Misses.Miss_Causes_A_Walk	0.5657
Dtlb_Load_Misses.Walk_Completed	0.5327
Dtlb_Load_Misses.Walk_Completed_4K	0.5226
BR_Inst_Retired.Cond.	0.4623
BR_Misp_Retired.All_Branch	0.4615
BR_Inst_Retired.Far_Branch	0.4608
BR_Misp_Exec.Taken_Cond.	0.4510
BR_Misp_Exec.Taken_Indirect_Jmp_Non_Call_Ret	0.4455
BR_Inst_Retired.Not_Taken	0.4412
BR_Misp_Retired.Cond.	0.3786
UOPS_Issued_Any	0.3663
BR_Misp_Exec.Nontaken_Cond.	0.3648
Dtlb_Load_Misses.Walk_STLB_Hit_4K	0.3627
BR_Inst_Exec.Taken_Direct_Near_Call	0.3618
BR_Inst_Exec.Taken_Indirect_Near_Call	0.3592
BR_Misp_Exec.Taken_Indirect_Near_Call	0.3553
BR_Misp_Exec.Taken_Ret_Near	0.3491
BR_Inst_Exec.Taken_Direct_Jmp	0.3455
BR_Inst_Exec.Taken_Cond.	0.3390
IDQ.Mite_UOPS	0.3383
BR_Inst_Exec.All_Direct_Jmp	0.3238
BR_Inst_Exec.Taken_Indirect_Jmp_Non_Call_Ret	0.3137
BR_Inst_Exec.Taken_Indirect_Near_Return	0.2944
BR_Misp_Retired.Near_Taken	0.2871
BR_Inst_Exec.Nontaken_Cond.	0.2703
BR_Misp_Exec.All_Branches	0.2700
BR_Inst_Exec.All_Cond.	0.2634
BR_Misp_Retired.All_Branches_Pebs	0.2111

Table A.7: Benchmark tests used in the experiments

Processor Tests			System Tests	Disk Tests	Memory Tests	Real-World	Attacks
1) Aobench	41) Minion 1	81) Graphics 1	120) Apache	153) Aio-stress	165) Mbw	174) Websites	1) Flush+Flush
2) Botan 1	42) Minion 2	82) Graphics 2	121) Battery	154) Blogbench 1	166) Ram 1	175) Videos	2) Flush+Reload
3) Botan 2	43) Minion 3	83) Graphics 3	122) Compress	155) Blogbench 2	167) Ram 2	176) MySQL	3) Prime+Probe
4) Botan 3	44) Perl 1	84) Graphics 4	123) Git	156) Compile	168) Ram 3	177) Apache	4) Meltdown
5) Botan 4	45) Perl 2	85) Graphics 5	124) Hint	157) Dbench	169) Ram 4	178) Office	5) Spectre
6) Botan 5	46) Radiance 1	86) Graphics 6	125) Nginx	158) Fio 1	170) Ram 5		6) Rowhammer
7) Bullet 1	47) Radiance 2	87) Graphics 7	126) Optcarrot	159) Fio 2	171) Stream		7) Zombieload
8) Bullet 2	48) Scimark 1	88) Hpcg	127) Php 1	160) Iozone	172) T-test		
9) Bullet 3	49) Scimark 2	89) Luajit 1	128) Php 2	161) Postmark	173) Tinymem		
10) Bullet 4	50) Scimark 3	90) Luajit 2	129) Pybench	162) Sqlite			
11) Bullet 5	51) Scimark 4	91) Luajit 3	130) Schbench	163) Tiobench			
12) Bullet 6	52) Scimark 5	92) Luajit 4	131) Stress-ng 1	164) Unpack			
13) Bullet 7	53) Scimark 6	93) Luajit 5	132) Stress-ng 2				
14) Cache 1	54) Swet	94) Luajit 6	133) Stress-ng 3				
15) Cache 2	55) Hackbench	95) Mencoder	134) Stress-ng 4				
16) Cache 3	56) M-queens	96) Multichase 1	135) Stress-ng 5				
17) Gzip	57) Mrbayes	97) Multichase 2	136) Stress-ng 6				
18) Dcraw	58) Npb 1	98) Multichase 3	137) Stress-ng 7				
19) Encode	59) Npb 2	99) Multichase 4	138) Stress-ng 8				
20) Ffmpeg	60) Npb 3	100) Multichase 5	139) Stress-ng 9				
21) Fhourstones	61) Npb 4	101) Polybench-c	140) Stress-ng 10				
22) Glibc 1	62) Npb 5	102) Sample	141) Stress-ng 11				
23) Glibc 2	63) Npb 6	103) Sudokut	142) Stress-ng 12				
24) Glibc 3	64) Npb 7	104) C-ray	143) Stress-ng 13				
25) Glibc 4	65) Povray	105) Cloverleaf	144) Stress-ng 14				
26) Glibc 5	66) Smallpt	106) Dacapo 1	145) Stress-ng 15				
27) Glibc 6	67) Tachyon	107) Dacapo 2	146) Stress-ng 16				
28) Glibc 7	68) Bork	108) Dacapo 3	147) Sunflow				
29) Glibc 8	69) Build Apache	109) Dacapo 4	148) Sysbench 1				
30) Gnupg	70) Byte 1	110) Dacapobench 5	149) Sysbench 2				
31) Java 1	71) Byte 2	111) John 1	150) Tensorflow				
32) Java 2	72) Byte 3	112) John 2	151) Tjbench				
33) Java 3	73) Byte 4	113) John 3	152) Xsbench				
34) Java 4	74) Clomp	114) Mafft					
35) Java 5	75) Crafty	115) N-queens					
36) Java 6	76) Dolfyn	116) Openssl					
37) Lzbenc 1	77) Espeak	117) Primesieve					
38) Lzbench 2	78) Fftw	118) Stockfish					
39) Lzbench 3	79) Gcrypt	119) Ttsiod					
40) Lzbench 4	80) Gmpbench						