Towards Security Analysis and Design of Confidential Computing Systems

Xiaoxuan Lou

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2024

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

26-May-2024

. . . .

Date

Xiaoxuan Lou

. . . .

Xiaoxuan Lou

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiargism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

26-May-2024
. . . .
Date

. . . .
Asst Prof Tianwei Zhang

Authorship Attribution Statement

This thesis contains materials from 3 papers published in the following peer-reviewed journal(s) / from papers accepted at conferences in which I am listed as an author.

Chapter 3 is published as Xiaoxuan Lou, Shangwei Guo, Jiwei Li, Yaoxin Wu, Tianwei Zhang, NASPY: Automated Extraction of Automated Machine Learning Models. in International Conference on Learning Representations, 2022. https://openreview.net/pdf?id=KhLK0sHMgXK.

The contributions of the co-authors are as follows:

- I was the lead author, I wrote the manuscript drafts and conducted all experiments.
- Prof. Tianwei Zhang guided the initial research direction and revised the manuscript drafts.
- I co-designed the methodology with Prof. Tianwei and Prof. Shangwei Guo.
- Prof. Jiwei Li, and Prof. Yaoxin Wu discussed and supported the research, and revised the drafts.

Chapter 4 is published as Xiaoxuan Lou, Kangjie Chen, Guowen Xu, Han Qiu, Shangwei Guo, Tianwei Zhang. Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. In Proceedings of the 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2024. https://dsn2024uq.github.io/cpaccepted.html.

The contributions of the co-authors are as follows:

- I was the lead author, I wrote the manuscript drafts and conducted all experiments.
- Prof. Tianwei Zhang guided the initial research direction and revised the manuscript drafts.
- I co-designed the methodology with Prof. Tianwei and Mr. Kangjie Chen.
- Dr. Guowen Xu, Prof. Han Qiu and Prof. Shangwei Guo discussed and supported the research, and revised the drafts.

Chapter 5 is published as Xiaoxuan Lou, Shangwei Guo, Jiwei Li, Tianwei Zhang. Ownership verification of dnn architectures via hardware cache side channels. In IEEE Transactions on Circuits and Systems for Video Technology, 2022. https://ieeexplore.ieee.org/abstract/document/9801864.
DOI: 10.1109/TCSVT.2022.3184644

The contributions of the co-authors are as follows:

- I was the lead author, I wrote the manuscript drafts and conducted all experiments.
- Prof. Tianwei Zhang guided the initial research direction and revised the manuscript drafts.
- I co-designed the methodology with Prof. Tianwei and Prof. Shangwei Guo.
- Prof. Jiwei Li discussed and supported the research, and revised the drafts.

26-May-2024

. . . .

Date

Xiaoxuan Lou

. . . .

Xiaoxuan Lou

Acknowledgements

It has been a while since I came to Singapore, this beautiful and welcoming country, to pursue further knowledge and realize my self-worth. Therefore, I would like to express my heartfelt gratitude to those who supported and encouraged me during my Ph.D. journey.

First and foremost, I express my most respectful thanks to my advisor, Prof. Tianwei Zhang. He is the best mentor throughout my research journey. Without his guidance, I would not have embarked on my Ph.D. program or learned how to navigate the path to becoming an independent researcher. His patience and profound academic expertise have contributed significantly to my growth as a researcher, instilling in me the skills and confidence to pursue independent research. His unwavering commitment to scientific rigor, keen insight into emerging fields, and boundless curiosity will undoubtedly continue to shape my academic journey in the future.

I am also grateful for those close colleagues in my life: Kangjie Chen, Gelei Deng, Dikai Liu, Wei Gao, Guanlin Li, Xingshuo Han, Qinghao Hu, Meng Zhang, Xiaobei Yan, Yutong Wu, Yi Xie, Zhaoxuan Wang, Meng Hao, Hanxiao Chen, Dr. Anran Li, Dr. Cen Zhang, Dr. Hu Ming, Dr. Jian Zhang. We work together in the same lab room, having a happy time with uncountable inspiring and exciting discussions together, and you help me a lot in my daily life.

I also extend my heartfelt thanks to Dr. Shangwei Guo and Dr. Guowen Xu, who help me a lot in my research works and I would always remember the time we spent working hard together.

All my roommates, Chengwei Liu, Shuo Sun, Wentao Zhang, are all my best friends. It is my great honor to have many precious memory with you guys and I hope we could all grow in our desired styles.

Lastly, but certainly not least, I want to express my deep appreciation to my parents. Without your support and understanding, I could not finish my Ph.D degree. Your unconditional love and unwavering encouragement are the driving force behind my continuous determination to move forward.

I want to express my heartfelt gratitude to everyone who has been mentioned, as well as those whose names may have been inadvertently left out. Your invaluable contributions to this chapter of my life have not gone unnoticed. The passion, love, and personal growth I have experienced during this journey have been made possible by each and every one of you. For this, I am forever indebted.

Contents

Acknowledgements ix

List of Figures xv

List of Tables xvii

Abstract 1

1 Introduction 3
- 1.1 Motivation . . . 5
- 1.2 Main Work . . . 6
- 1.3 Contribution of the Thesis . . . 8
- 1.4 List of Materials Related to the Thesis . . . 9
- 1.5 Outline of the Thesis . . . 10

2 Related Works 11
- 2.1 Micro-architectural Side-channel Studies . . . 11
  - 2.1.1 Side-channel Attacks . . . 11
  - 2.1.2 Defenses against Side Channels . . . 14
- 2.2 Novel Confidential Computing System Designs . . . 15
  - 2.2.1 Confidential Machine Learning (ML) . . . 15
  - 2.2.2 Confidential Distributed Computing . . . 17

I New Side-channel Investigation in Confidential Computing 19

3 Automated Extraction of Automated Machine Learning Models 21
- 3.1 Introduction . . . 21
- 3.2 Background . . . 24
  - 3.2.1 Neural Architecture Search (NAS) . . . 24
  - 3.2.2 Hardware attacks . . . 25
  - 3.2.3 Sequence-to-sequence learning . . . 26
- 3.3 Framework Overview . . . 26

3.3.1 Threat Model . . . 26
3.3.2 Attack Overview . . . 27
3.4 Detailed Design . . . 28
3.4.1 Operation Sequence Identification . . . 28
3.4.2 Hyper-parameter Recovery . . . 32
3.4.3 Model Topology Reconstruction . . . 33
3.5 Evaluation . . . 34
3.5.1 Operation Sequence Identification . . . 35
3.5.2 Hyper-parameter Recovery . . . 37
3.5.3 Model Topology Reconstruction . . . 38
3.6 Discussions . . . 39
3.7 Conclusion . . . 40

4 Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels 41

4.1 Introduction . . . 42
4.2 Background and Related Works . . . 45
4.2.1 Hardware Performance Counters . . . 45
4.2.2 Secure Encrypted Virtualization . . . 45
4.2.3 Fuzzing . . . 46
4.2.4 Differential Privacy . . . 46
4.3 HPC Side Channels . . . 47
4.3.1 Threat Model . . . 47
4.3.2 Abstraction of HPC Side-channel Attacks . . . 48
4.3.3 Website Fingerprinting Attack . . . 48
4.3.4 Keystroke Sniffing Attack . . . 49
4.3.5 Model Extraction Attack . . . 50
4.4 Framework Overview . . . 51
4.5 Application Profiler . . . 52
4.5.1 Challenges . . . 52
4.5.2 Profiling Design . . . 53
4.6 Event Fuzzer . . . 56
4.6.1 Challenges . . . 57
4.6.2 Design Overview . . . 57
4.6.3 Instruction Cleanup . . . 58
4.6.4 Code Generation and Execution . . . 59
4.6.5 Result Confirmation . . . 60
4.6.6 Gadgets Filtering . . . 61
4.7 Event Obfuscator . . . 62
4.7.1 Challenges and Insight . . . 62
4.7.2 Differential Privacy Mechanisms . . . 63
4.7.3 Design Details . . . 64
4.8 Evaluation . . . 66

4.8.1 Profiling Evaluation . . . 66
4.8.2 Fuzzing Evaluation . . . 67
4.8.3 Defense Effectiveness . . . 68
4.8.4 Defense Efficiency . . . 70
4.9 Discussion . . . 71
4.9.1 Alternative Defense Strategies . . . 71
4.9.2 Analysis with Multiple Tries . . . 73
4.10 Conclusion . . . 73

II New Designs for Confidential Computing with Emerging Applications 75

5 Ownership Verification of DNN Architectures via Hardware Cache Side Channels 77
5.1 Introduction . . . 78
5.2 Related Works on DNN Watermarking . . . 81
5.2.1 White-box solutions . . . 81
5.2.2 Black-box solutions . . . 81
5.3 Preliminaries . . . 82
5.3.1 Definition of A NAS Method . . . 82
5.3.2 Definition of A Watermarking Scheme . . . 82
5.4 My Watermarking Scheme . . . 84
5.4.1 Watermark Generation (WMGen) . . . 84
5.4.2 Watermark Embedding (Mark) . . . 86
5.4.3 Watermark Verification (Verify) . . . 87
5.4.4 Theoretical Analysis . . . 88
5.5 Side Channel Extraction . . . 88
5.5.1 Method Overview . . . 89
5.5.2 Recovery of NAS Operations . . . 90
5.6 Evaluation . . . 94
5.6.1 Experimental Setup . . . 94
5.6.2 Effectiveness . . . 95
5.6.2.1 Key Generation . . . 95
5.6.2.2 Watermark Embedding . . . 95
5.6.2.3 Watermark Extraction and Verification . . . 96
5.6.3 Usability . . . 98
5.6.4 Robustness . . . 99
5.6.5 Uniqueness . . . 103
5.7 Conclusion . . . 105

6 Enabling Fast and Secure Function Cold Starts in Confidential Serverless Systems 107
6.1 Introduction . . . 108

6.2 Background . . . 111
6.2.1 Confidential Serverless Computing . . . 111
6.2.2 AMD SEV . . . 112
6.2.3 Unikernel . . . 113
6.3 Motivation . . . 113
6.3.1 Startup Latency of SEV VM . . . 113
6.3.2 Analysis of Startup Optimizations . . . 115
6.4 System Overview . . . 118
6.4.1 Threat Model . . . 118
6.4.2 Design Principles & Challenges . . . 119
6.4.3 System Architecture and Workflow . . . 119
6.5 Guest Unikernel Analysis . . . 121
6.5.1 Adaptations for Confidential Serverless . . . 121
6.5.2 Memory Layout Analysis . . . 122
6.6 Neuralyzer Layer Design . . . 124
6.6.1 Fix Memory Access Permission . . . 124
6.6.2 Restore Module . . . 125
6.6.3 Attestation Module . . . 127
6.7 Evaluation . . . 128
6.7.1 Experimental Setup . . . 128
6.7.2 Startup Latency . . . 129
6.7.3 Memory Overhead . . . 131
6.7.4 End-to-End Performance . . . 131
6.7.5 Overhead of the Initial Booting . . . 133
6.8 Security Analysis . . . 134
6.9 Conclusion . . . 136

7 Conclusion and Future Work **137**
7.1 Conclusion . . . 137
7.2 Future Work . . . 139

Bibliography **143**

List of Figures

1.1 Architecture of confidential computing system. 4
1.2 Typical application framework for confidential computing. 5
1.3 Structure of my research works. 7

3.1 Architecture of a NAS model based on cells 25
3.2 Overview of threat model. 27
3.3 Workflow of my model extraction framework. 28
3.4 GEMM procedure. 29
3.5 Event sequences of four representative operations in NAS models. . 30
3.6 Procedure of operation sequence identification. 32
3.7 (a) Average OER of the two identification models with different configurations. (b) Loss and OER trend of the RNN-CTC model. (c) OER of the transformer model on validation samples 35
3.8 Inter-operation context testing. (a) OER trend of RNN-CTC. (2) OER of the transformer on validation samples. 37
3.9 Robustness versus different scales of noise. (a) OER of RNN-CTC. (b) OER of the transformer. 37
3.10 Recovery accuracy. 37
3.11 Memory address trace of a NAS cell. 37

4.1 Training curves of three HPC side-channel attacks. 49
4.2 Overview of my `Aegis` framework. 51
4.3 The distribution of HPC event values. 56
4.4 State transition. 58
4.5 Workflow of Event Fuzzer. 59
4.6 Execution of repeated triggers. 61
4.7 Workflow of Event Obfuscator. 65
4.8 The mutual information of each HPC event. 67
4.9 Impact of ϵ on various attacks. 69
4.10 Impact of ϵ on the latency overhead (upper) and CPU usage (lower). 70
4.11 Attack accuracy with the random noise 72

5.1 Overview of my watermarking framework 84
5.2 Event sequences of four representative operations in NAS models. . 89
5.3 Implementing a convolution operation as matrix multiplication . . . 91
5.4 Procedure of separable convolutions. 92

5.5 Architectures of the searched cells. c_{i-1} and c_{i-2} are the inputs from the previous cells. 96
5.6 A side-channel trace of the first normal cell. 97
5.7 Execution time of the operations in a cell 97
5.8 Extracted values of the matrix parameters (m, n, k). 98
5.9 Top-1 validation accuracy 99
5.10 Side-channel traces of weighting pruned models. 101
5.11 Traces of obfuscated models. 102
5.12 Influence of useless cell windows. 102
5.13 Influence of parameter binarization. 103
5.14 Operation distributions for a normal cell (left) and reduction cell (right). The connection index is the index of the connection edge in the NAS cell. 104

6.1 Architecture of confidential serverless computing 111
6.2 Fields of an RMP entry [1, Table 15-36] 112
6.3 Boot process of normal VM and SEV VM in serverless computing. . 113
6.4 Startup latency of the SEV VM (left bars) and the normal VM (right bars), and SGX (line, sourced from [2]). 115
6.5 Caching performance of serverless functions. 117
6.6 Performance of Save/Restore method. 117
6.7 Overview of Neuralyzer architecture and workflow 120
6.8 Memory layout of the guest unikernel 123
6.9 Workflow of restore module 126
6.10 Workflow of attestation module 127
6.11 Comparison with three baselines 130
6.12 Time breakdown of end-to-end function execution on Neuralyzer (left bars) and native SEV VM(right bars). 133
6.13 Initial booting performance 134

List of Tables

2.1 Caption for LOF 12

3.1 Testing OER for three NAS methods. 36
3.2 Accuracy (%) of random models on two datasets. 38

4.1 Statistics of HPC events in various processors. 53
4.2 HPC event distribution, including events of Hardware (H), Software (S), Hardware Cache (HC), Tracepoint (T), Raw CPU (R) and Others (O). Data in brackets shows the percentage remaining after the warm-up profiling. 54
4.3 Time consumption for each fuzzing step. 68

5.1 An example of the marking key *mk*. 95
5.2 Accuracy of structured pruned models on CIFAR10 103

6.1 Evaluations of existing optimizations. ⧫✓ denotes the feature is enabled if future TEE hardware supports Fork. 116
6.2 Serverless workload functions from FunctionBench 131
6.3 Time breakdown of end-to-end function execution, including Request Relay (RR), Restore/Launch (R/L), Generate Report (GR), Verify Signature (VS) and Execution (EXE). 132

Abstract

Confidential computing has emerged as a critical security technology for addressing user privacy issues in cloud computing, which is also a hot subject in contemporary security technologies. By leveraging collaborative security in both hardware and software, it establishes an encrypted Trusted Execution Environment (TEE) to ensure confidentiality and integrity protection for data in use. The introduction of confidential computing provides the final aspect of security assurance across the entire data usage pipeline, encompassing three states: in transit, at rest, and in use. While existing confidential computing systems can mitigate a majority of software and hardware attacks, including privileged hypervisor attacks and cold boot attacks, they still suffer from two significant problems: (1) Vulnerability to micro-architectural side-channel attacks, which leverage side-channel information to reveal secrets sealed within the encrypted TEE sandbox; (2) Security and efficiency issues when handling emerging applications, like machine learning and serverless computing. These two concerns restrict the widespread adoption and further development of confidential computing.

For the first problem, I conduct a comprehensive security analysis of existing confidential computing systems, aiming to identify novel side-channel attack vectors that can breach the TEE isolation and propose unified defense frameworks for those unexplored side-channel leakages. I first propose `NASPY`, an end-to-end attack method that can reveal novel Neural Architecture Search (NAS) models from the encrypted TEE black box. It shows the possibility of breaching confidential computing systems and stealing sealed sensitive information. After that, `Aegis` is proposed as a unified defense framework for mitigating confidential virtual machines from Hardware Performance Counter (HPC) side channels. This work prevents the leakage source effectively and efficiently.

For the second problem, I design more novel confidential computing systems that integrate with recently emerging workloads, including machine learning and serverless computing. I first design a watermarking scheme for verifying the ownership

of deep learning models sealed inside the TEE sandbox. It enhances the security guarantee of confidential AI systems. Furthermore, I also integrate confidential computing with serverless computing to design a novel fast-launched confidential serverless computing system. This system ensures the confidentiality of user data while retaining the performance advantages offered by serverless computing.

In summary, this thesis is dedicated to analyzing the security of confidential computing systems, identifying new attack vectors, presenting a unified defense framework, and designing novel systems integrating emerging workloads.

Chapter 1

Introduction

In recent decades, the rapid advancement of cloud computing has propelled its widespread adoption across a multitude of applications. As cloud computing ecosystems become mature, there is a heightened focus on ensuring robust security and privacy guarantees. These considerations have emerged as top priorities alongside performance efficiency for both cloud service providers and customers. The Confidential Computing Consortium defines data security from three states: in transit, at rest and in use [3]. While data existing in the first two states have been well protected with various mechanisms, e.g., HTTPS and disk encryption, data in use is still exposed to threat from attackers, especially those potential malicious cloud hypervisors who own the highest system privileges. Motivated by this concern, *Confidential Computing* is proposed to protect data in use by performing computations in a hardware-based, attested Trusted Execution Environment (TEE). As a sophisticated and innovative technology, TEE establishes an exclusive and encrypted memory area dedicated to a specified sensitive program. This ensures that all operations occur solely within this isolated environment, preventing data leakage to external entities.

Existing TEE mechanisms can be divided into two classes: process-level TEE and system-level TEE. The former provides user-level secure enclaves for specific processes through hardware instructions, represented by Intel SGX [4], while the latter combines the isolation mechanism of virtualization technology to build a TEE with virtual machines as nodes, represented by AMD SEV [5] and Intel TDX [6].

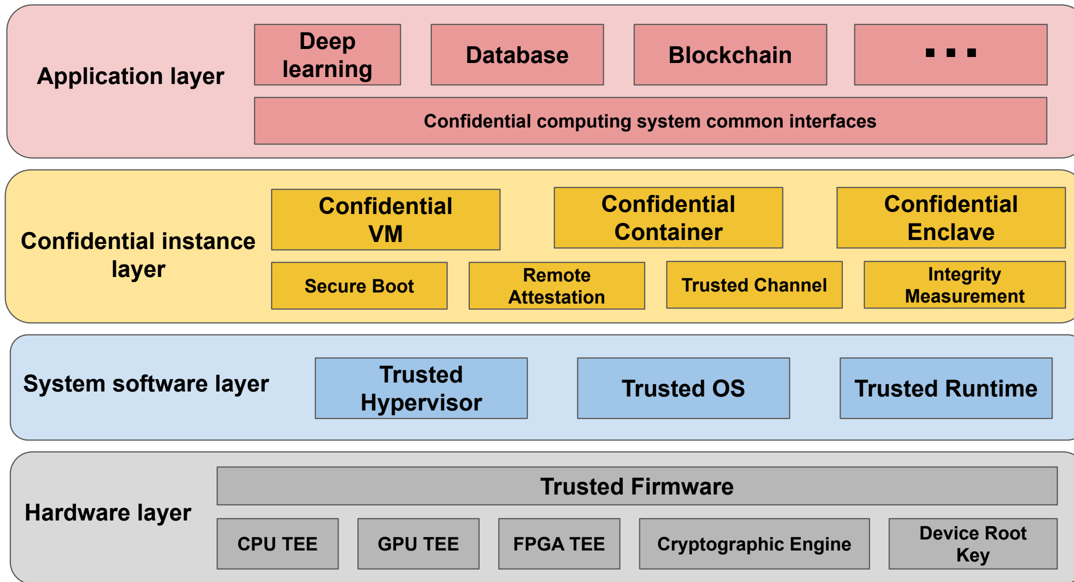


FIGURE 1.1: Architecture of confidential computing system.

In practical applications, a confidential computing system represents a collaborative integration of hardware and software trust technologies, as shown in Figure 1.1. The system can be divided into four levels:

(1) Hardware layer: this layer provides the root of trust for the entire confidential computing system and is usually bound with specific hardware processor vendors, including CPU vendors like Intel, AMD and ARM or GPU vendors like Nvidia. It delivers necessary hardware security primitives, e.g., encryption engine for encrypting memory area, and also the initial environment security boot.

(2) System software layer: this layer mainly denotes the operating system or privileged runtime that manages the hardware resources for upper layer applications, including the isolation of software components and secure context switch.

(3) Confidential instance layer: this layer is the kernel component of confidential computing system, which is also the primary area of focus and interaction for developers. Based on the TEE type it utilizes, the instance adopted in this layer can be confidential VM/container for system-level TEE, or confidential enclave for process-level TEE.

(4) Application layer: this layer is the interface communicating with users and customers. With the use of system-level TEE, the application in this layer can

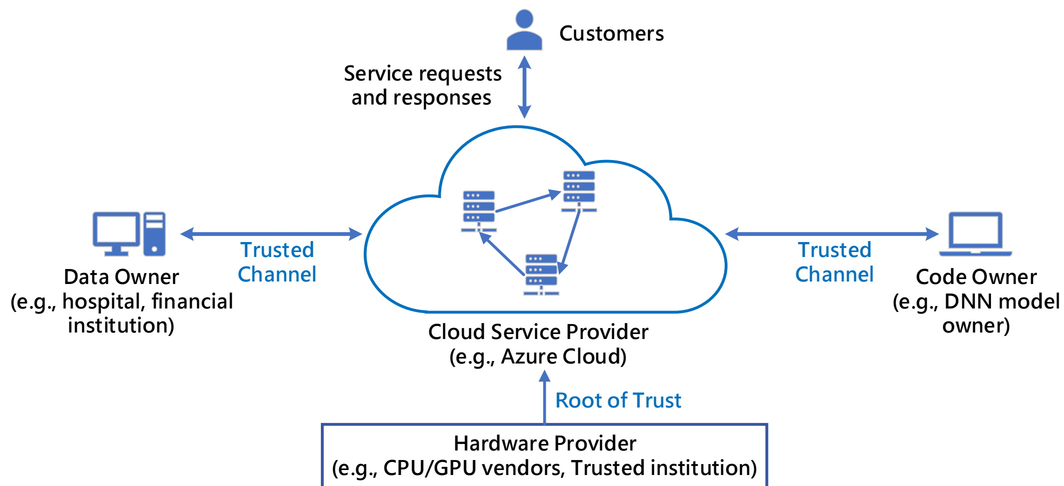


FIGURE 1.2: Typical application framework for confidential computing.

be deployed without any modification. However, for process-level TEE, the application is required to undergo specific modifications to accommodate the unique programming model.

Based on the above system architecture, a typical application framework for confidential computing is shown in Figure 1.2. It mainly involves five participants roles: hardware provider, cloud service provider, data owner, code owner and customers. The sensitive data and code can be uploaded through the trusted channel and then deployed as a confidential instance that is isolated from the possible malicious cloud service provider. The security and confidentiality of the entire system is guaranteed by the trusted hardware provider. Finally, customers can directly communicate with the target service using conventional requests and responses.

1.1 Motivation

While TEE-based confidential computing systems are capable of thwarting nearly all hardware attacks and privileged software attacks, they still face two significant challenges: (1) Vulnerability to *micro-architectural side-channel attacks*, which use side-channel information (e.g., execution time, memory footprints) to infer the secrets in the encrypted environment; (2) Narrow scope of explored confidential workloads, leading to suboptimal performance and incompatibility with emerging usage scenarios.

For the first challenge, although past efforts have been devoted to reinforce confidential computing systems against different micro-architectural side channels, e.g., CPU cache [7, 8], page faults [9, 10], branch prediction [11, 12], new attack methods still continue to emerge incessantly and bypass existing defenses. Furthermore, some side-channel leakage sources (e.g., hardware performance counters) have not been systematically studied for this secnario. Hence, it motivates us to perform further security analysis on existing confidential computing systems, aiming to **investigate novel attack vectors and propose unified defense frameworks**, thus enhancing security guarantee provided by confidential instances.

For the second challenge, given confidential computing is still in its infancy stage, researchers mainly focus on improving its security guarantee while neglecting novel system designs for emerging workloads, like machine learning models and serverless cloud computing. Some previous works have noted this issue, integrating TEE technologies to propose confidential machine learning systems [13–19] and confidential serverless computing systems [2, 20–25]. However, these works expose various security and efficiency issues, which motivate us to **design novel confidential computing systems that are compatible with emerging workloads**, thus further expanding the influence and application scope of trusted computing.

1.2 Main Work

To solve the problems of current confidential computing systems mentioned above, I propose four research works, whose logical connections are shown in Figure 1.3. They can be divided into two directions. First, I conduct an in-depth security analysis on the existing confidential computing systems, from the perspectives of novel attacks and defenses. Furthermore, I design more effective and efficient confidential computing systems based on existing hardware TEEs for recently emerging workloads, including the DNN model watermark and efficient serverless computing. My four pioneering works are outlined below:

Automated Extraction of Automated Machine Learning Models. I introduce `NASPY`, an end-to-end adversarial framework to extract deep learning model architectures from a sealed TEE sandbox with analyzing side-channel leakages. While previous

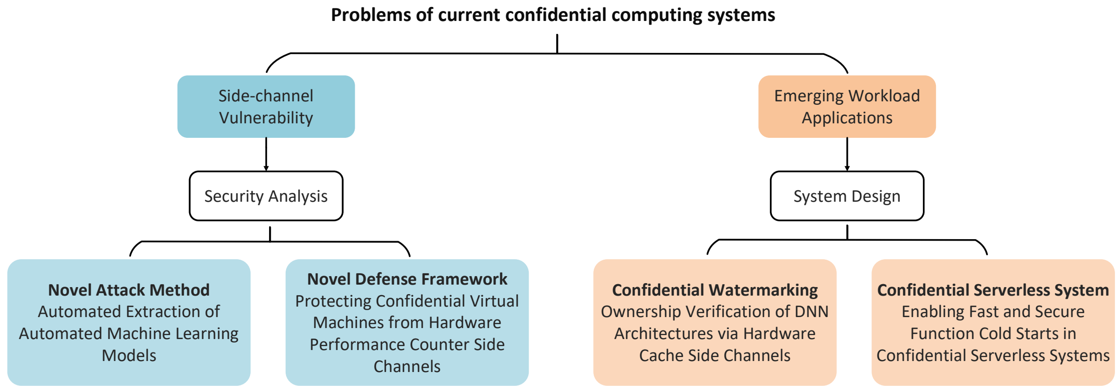


FIGURE 1.3: Structure of my research works.

extraction attacks mainly focus on conventional Deep Neural Network (DNN) models with very simple operations, `NASPY` targets on the novel deep learning models generated by Neural Architecture Search (NAS). I introduces seq2seq models to automatically identify novel and complicated operations (e.g., separable convolution, dilated convolution) from hardware side-channel sequences of model inference, which bypasses the heavy manual analysis with lots of prior knowledge. As a result, `NASPY` is able to extract the complete NAS model architecture with high fidelity and automation, where the error rate can be reduced to only 3.2%.

Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. While many studies have focused on mitigating micro-architectural side channels in confidential computing systems, the potential threat posed by Hardware Performance Counter (HPC) side channels has been largely overlooked. Previous research has failed to address this prominent source of information leakage, highlighting a gap in the existing defenses against such side-channel vulnerabilities. Hence, I introduce `Aegis`, a unified framework for defending confidential virtual machines against HPC side channels with provable privacy guarantee and minimal performance overhead. I present three case studies to demonstrate that `Aegis` can defeat different types of HPC side-channel attacks (i.e., website fingerprinting, DNN model extraction, keystroke sniffing). Evaluations show that `Aegis` can effectively decrease the attack accuracy from 90% to 2%, with only 3% overhead on the application execution time and 7% overhead on the CPU usage.

Ownership Verification of DNN Architectures via Hardware Cache Side Channels. Sealing Deep Neural Network (DNN) models inside a TEE sandbox is a double-edged sword, which not only enables IP security for customers but also allows the

attacker to hide their stolen models from ownership verification and inspection. Hence, I present a novel watermarking scheme, which can verify the ownership of DNN models by analyzing the side-channel leakages. This scheme can identify the DNN model architectures with high fidelity even when they are sealed inside an encrypted sandbox. I conduct comprehensive experiments on watermarked CNN models for image classification tasks and the experimental results show my scheme has negligible impact on the model performance, and exhibits strong robustness against various model transformations and adaptive attacks.

Enabling Fast and Secure Function Cold Starts in Confidential Serverless Systems. As the next stage in cloud computing, serverless computing has garnered significant attention in recent years, which also makes confidential serverless computing a focus of both industry and academia. However, integrating TEEs with serverless computing leads to a significant startup latency of function executors. To address this issue, I propose `Neuralyzer`, a confidential serverless system that ensures robust security of guest users while retaining the performance improvement offered by cached executors. My comprehensive experiments on three baselines and six real-world serverless functions show that `Neuralyzer` system can dramatically reduce the startup latency by 76$\sim$ 501$\times$ compared with native confidential VMs, and achieve an end-to-end service latency of only hundreds of milliseconds.

1.3 Contribution of the Thesis

This thesis makes several significant contributions to the field of confidential computing systems:

1. Novel DNN Extraction Attack Targeting Sealed TEE Sandbox. I develop a novel attack vector that leverages cache side channels from the TEE sandbox to extract DNN model architectures even when it is sealed inside the encrypted environment. It is the first work that achieves automatically extracting novel NAS models from the black-box TEE instances.

2. Unified Defense Framework against HPC Side Channels. I propose a unified defense framework for confidential virtual machines to prevent HPC side channels, which are considered as the easiest-to-use leakage source but have not been given due attention. It is the first work that systematically studies the HPC

side channels and achieves preventing such side channels with small performance overhead.

3. Novel DNN Model Ownership Verification Scheme. In the context of confidential machine learning protected by TEE sandboxes, I propose a novel watermarking scheme for verifying the IP ownership of a DNN model. It is the first design that integrates side-channel analysis with confidential machine learning to enhance security assurance of IP models.

4. Novel Confidential Serverless System with Superior Performance. While conventional confidential serverless computing typically results in a significant slowdown (more than 1000×), I propose a novel confidential serverless system `Neuralyzer` to bridge this performance gap. It achieves small startup latency in mere milliseconds, making it the first work that ensures both security guarantee and performance advantage for serverless computing.

An in-depth security analysis on existing confidential computing system yields valuable insights, motivating the above research works. Overall, this thesis serves multiple vital purposes: (1) Advanced attack vectors: By delving into current confidential computing systems, I discovered new advanced attack vectors that can breach the security guarantee, which help researchers to enhance future system designs. (2) Unified defense frameworks: I also presented a unified defense framework to enhance confidential computing systems against side-channel leakages, providing a possible direction for future works. (3) Novel system designs: To prompt the practical adoption of confidential computing in real-world applications , I proposed novel system designs for integrating confidential computing with emerging workloads, like the DNN models and serverless computing, with the aim of addressing both security and performance concerns.

1.4 List of Materials Related to the Thesis

The thesis mainly contains the materials from the following papers.

- **Xiaoxuan Lou**, Shangwei Guo, Jiwei Li, Yaoxin Wu, Tianwei Zhang, NASPY: Automated Extraction of Automated Machine Learning Models. in *International Conference on Learning Representations*, 2022.

- **Xiaoxuan Lou**, Kangjie Chen, Guowen Xu, Han Qiu, Shangwei Guo, Tianwei Zhang. Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. In *Proceedings of the 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2024.

- **Xiaoxuan Lou**, Shangwei Guo, Jiwei Li, Tianwei Zhang. Ownership verification of dnn architectures via hardware cache side channels. In *IEEE Transactions on Circuits and Systems for Video Technology*, 2022.

1.5 Outline of the Thesis

The remainder of this thesis is organized as follows:

Chapter 1 provides an overview of this thesis, as well as the motivations, main work, and contributions of the thesis.

Chapter 2 reviews the related works.

Chapter 3 proposes a novel DNN model extraction attack from a TEE-sealed black-box environment.

Chapter 4 presents a unified framework to protect confidential virtual machines from hardware performance counter side channels.

Chapter 5 combines cache side channels with TEE sandboxes to design a novel DNN model watermark for performing ownership verification.

Chapter 6 integrates TEE executors with serverless computing to design a novel efficient confidential serverless computing system, which achieves fast execution with guaranteed security.

Chapter 7 concludes above research works and give a discussion about possible future directions.

Chapter 2

Related Works

The significant threat posed by micro-architectural side channels is among the most crucial and active research in confidential computing systems. Additionally, the design of more innovative, secure, and widely applicable confidential computing systems is equally paramount. In this chapter, I give a comprehensive literature review on the state-of-the-art works related to the side-channel attacks and defenses, and the latest novel designs for confidential computing systems with emerging applications.

2.1 Micro-architectural Side-channel Studies

2.1.1 Side-channel Attacks

The history of side-channel attacks dates back to the year of 1996, when Kocher [69] demonstrated that the data leaked from timing channels was sufficient for an attacker to recover the entire secret key. To generalize, vulnerable implementations of system operations can exhibit secret-dependent non-functional behaviors during the time of execution, which an adversary can observe and utilize to fully or partially recover sensitive information. Since then, numerous types of side channels (e.g., execution timing [70, 71], acoustic emission [72], electromagnetic radiation [73] and power consumption [74]) have been discovered and exploited to defeat modern cryptographic schemes, allowing adversaries to break strong ciphers in a

Level	Category	Sharing	Attacks	Requirements
Instruction	Multiply	■	Multiplier unit contention [26, 27]	
	Floating point	■	FPU contention [28]	
	Branch	■	BTB contention [11, 29]	[11] requires Intel SGX
	Micro-operation	■	Port contention [30]	
Cache	Cache set	■: L1 & L2, ●: LLC	PRIME-PROBE [31–48] EVICT-TIME [31], PRIME-ABORT [49]	[39–41, 46–48] require Intel SGX [49] requires Intel TSX
	Cache line	■: L1 & L2, ●: LLC	FLUSH-RELOAD [50–54] FLUSH-FLUSH [55], RELOAD+REFRESH [56] COLLIDE-PROBE, LOAD-RELOAD[57] LRU state leaking[58]	Requires KSM [57] requires AMD predictor
	Cache bank	■	Bank contention [59], MemJam [60]	[60] requires Intel SGX
Memory Page	Page	■	TLB contention [61, 62]	
		▲	Page Fault/Table Entry [63–66]	Requires Intel SGX
	DRAM bank row	▲	Row buffer contention [67] Rambleed [68]	

TABLE 2.1: Side-channel attack vectors in hardware. ■: sharing the same CPU core; ●: sharing the same package. ▲: sharing the same computer.

short period of time with very few trials. In recent decades, the focus of various side-channel attacks has gradually shifted from cryptographic secrets to operating system security. Among these side-channel threats, micro-architectural attacks are particularly dangerous and prevalent. For example, the proposing of Meltdown [75] and Spectre [76] leads to a significant upheaval in the system security field.

A fundamental cause of such attacks is the conflict between *performance* and *security*. During the evolution of computer architecture, various strategies were introduced to speed up the execution, which may bring side channels that leak the information of applications running on the system. Table 2.1 characterizes the attack vectors of side-channel techniques based on different levels of the computer system. One representative example is **caching**: a small hardware component is introduced (e.g., CPU caches, Translation Look-aside Buffer, DRAM row buffer) to store the previously accessed data, which is usually expected to be used again soon due to the principle of locality. Fetching data directly from this component is much faster. However, such timing differences can reveal the victim program's access traces [31, 33, 61]. A quantity of techniques have been designed over the past decades to realize cache side-channel attacks. Two representative attacks are described as below. (1) In a PRIME-PROBE attack [77], the adversary first fills up the critical cache sets with its own memory lines. Then the victim executes and potentially evicts the adversary's data out of the cache. After that, the adversary measures the access time of each memory line loaded previously. A longer access time indicates that the victim used the corresponding cache set. (2) A FLUSH-RELOAD attack [78] requires the adversary to share the critical memory lines with

the victim, e.g., via shared library. The adversary first evicts these memory lines out of the cache using dedicated instructions (e.g., *clflush*). After a period of time, it reloads the lines into the cache and measures the access time. A shorter time indicates the lines were accessed by the victim.

Micro-architectural side-channel attacks are also used for revealing certain emerging workloads, like deep learning models. Hong et al. [79] recovered the architecture attributes by observing the caching invocations of critical functions in the deep learning frameworks (e.g., Pytorch, TensorFlow). Yan et al. [80] proposed Cache Telepathy, which monitors the low-level BLAS library to achieve stealing deep learning model architectures. Some works leveraged other side channels to extract DNN models. Batina et al. [81] extracted a functionally equivalent model by monitoring the electromagnetic signals of a microprocessor hosting the inference program. Duddu et al. [82] found that models with different depths have different execution time, which can be used as a timing channel to leak the network details. Memory side-channels were discovered to infer the network structure of DNN models on GPUs [83] and DNN accelerators [84].

Given the design model of TEE-based confidential computing systems largely ignores side-channel attacks, those conventional side-channel techniques actually can be directly applied to steal secrets from the protected TEE sandbox. Even the latest TEE designs, e.g., AMD SEV-SNP [85] and Intel TDX [6], have contained mitigation for certain side channels, they still just cover a tiny range of potential attack surface. More seriously, given TEE usually assumes that the OS is not trusted, it is also a common way to use page tables and their attributes to conduct side-channel attacks, typically represented by controlled-channel attacks [63] and SGX-PTE [66]. This provides the attacker much stronger capabilities than attacker in the conventional threat models. Previous works have shown that, if the attacker is the malicious OS, he can obtain fine-grained information in an easier way by manipulating the OS interrupt (e.g., SGX-Step [86]). If the attacker is a normal user, he can also use enclaves to hide malicious behaviors [48]. In recent years, a variety of side-channel attacks have been proposed to breach the security guarantee of existing confidential computing systems, which establish side channels via performance counters [87], cache occupancy [77], unprotected I/O operations [88, 89], page faults [90–92], etc. Hence, it is an urgent demand to perform a security analysis on existing confidential computing systems and identify potential

side-channel attack vectors.

2.1.2 Defenses against Side Channels

Compared to the constant emergence of attack methods, side-channel defenses are relatively scarce, with users often doubting their feasibility and efficiency. The root cause of side-channel attacks is the sharing of certain system resources, which is also the focus of designing effective defenses. However, it is obviously infeasible to disable those features for side-channel mitigation, which can incur tremendous performance overhead. Therefore, effective elimination of side-channel vulnerabilities has been a long-standing goal. There are three types of classic system-level strategies for defending against side-channel attacks:

Process execution partitioning. From an operating system perspective, prohibiting the sharing of sensitive system resources among different users shows promise as a defense method. The first strategy is spatial partitioning, i.e., assigning different parts of the hardware units to processes. For instance, in the cloud scenario, hypervisor-based solutions were designed to defeat Last Level Cache (LLC) attacks by partitioning the LLC, via page coloring [93], page locking [94], and Intel Cache Allocation Technology [93].

Process scheduling. Another possible strategy is to carefully schedule different programs to achieve temporal partitioning, so that the attacker cannot run concurrently with the victim on the same machine. Zhang and Reiter [95] introduced an OS-based solution, which frequently flushes the local microarchitectural states (BTB, TLB, caches) to reduce side-channel leakage during context switches. Similar ideas were proposed in [96, 97], where CPU caches are flushed during VM switches to defeat cache side-channel attacks in the cloud. To reduce the overhead of state cleansing operations, Sprabery et al. [98] implemented the scheduling as an extension to the Completely-Fair-Scheduler in Linux.

Measurement randomization. The final strategy is to randomize the measurement of side-channel leakages. making it difficult or infeasible to capture accurate information based on the observations. This was first proposed in [99] to fuzz the timing information to reduce timing channels. Vattikonda et al. [100] modified the *rdtsc* instruction from the hypervisor to randomize the emulated timer. Martin

et al. [101] optimized this approach by adding random noise in each predefined epoch. Li et al. [102] introduced Stopwatch, which disables precise timing measurement in the cloud server to mitigate timing-channel attacks. It can also add randomization inside the application during compiling. Crane et al. [103] designed an approach to dynamically randomize the control flow in the application to defeat cache side-channel attacks. Braun et al. [104] inserted random temporal paddings into the source application to obfuscate the adversary's observations.

However, most of above defenses are unavailable for confidential computing systems, where the privileged operating system itself is a potential malicious attacker. To bridge this gap, past efforts have been devoted to reinforce the confidential computing systems against different micro-architectural side channels, e.g., CPU cache [7, 8], page faults [9, 10], branch prediction [11, 12]. Obviously, these efforts pale in comparison to the onslaught of incoming tremendous attacks. Hence, proposing new defenses to mitigate attacks more effectively and efficiently is important.

Overall, although security-aware systems and architectures were designed to mitigate side-channel attacks, it is however still very challenging to remove all side-channel vulnerabilities from the software implementations and hardware designs. As such, the arms race between side-channel attacks and defenses remains heated.

2.2 Novel Confidential Computing System Designs

Currently, the advancements in confidential computing systems have been increasingly applied across various industries, including finance, government, and healthcare. Of particular note are the applications of confidential machine learning and confidential distributed computing.

2.2.1 Confidential Machine Learning (ML)

With the advancement and widespread adoption of artificial intelligence technology, concerns regarding its data privacy and security have become prominent within the industry. The attack surface across the entire pipeline of machine learning is

extensive, presenting significant threats to the data security of both model users and owners. These threats include data poisoning attacks [105–108], data reconstruction attacks [109–112] and model extraction attacks [83, 113, 114]. Leveraging TEE sandbox to safeguard segments of the ML pipeline represents a leading edge in AI security research. Presently, research efforts focus on the use of confidential computing to protect the ML process, with a primary interest in securing the model training and inference processes during deployment. These works can be primarily categorized into two classes:

Complete model sealing. Citadel [13] exemplifies a comprehensive approach to safeguarding the entire pipeline for model training and inference within a TEE sandbox, ensuring the confidentiality of both models and data. DarkneTZ [14] serves as a notable scheme for protecting machine learning models using TEE, with a focus on deploying trained models to edge devices. PPFL [15] stands out as a typical scheme for ongoing protection of ML models, particularly aimed at preserving data confidentiality in horizontal federated learning scenarios. GradSec [16] introduces an enhanced approach that dynamically shields sensitive layers of ML models, thereby reducing the Trusted Computing Base (TCB) size and overall training time.

Partial model outsourcing. SLALOM [17] represents a noteworthy scheme for outsourcing computation protection, leveraging matrix multiplication linearity to incorporate blinding factors into data before outsourcing it to an untrusted GPU. DarKnight [18] builds upon SLALOM with improvements such as employing multiple GPUs to blind input data through linear combination with batch and random noise, and employing redundant GPUs to verify computation results, thereby ensuring the integrity and confidentiality of deep neural network (DNN) training. Goten [19] is a scheme that utilizes secret sharing to outsource the computation of linear layers based on a multiplicative secret sharing protocol among three untrusted GPUs. It leverages TEE to pre-share the random number seed, thereby reducing the number of negotiation rounds for the secret sharing protocol.

2.2.2 Confidential Distributed Computing

In addition to machine learning systems, confidential computing also finds a wide range of applications in distributed computing domains such as serverless computing, function encryption, blockchain, databases, and more.

IRON [115] introduces confidential computing to implement the first practical and provably secure functional encryption system that can be instantiated on real trusted hardware. EnclaveDB [116] leverages confidential computing technology to enhance database security by providing confidentiality, integrity, and freshness guarantees for both data and data queries. Other applications of confidential databases include Microsoft Azure Always Encrypted (AE) [117], Aliyun Operon [118], Huawei GaussDB [119]. Ekiden [120] integrates confidential computing with smart contracts to devise a novel architecture that separates *consensus* and *execution*.

In this thesis, I mainly focus on the latest emerging distributed computing workload, i.e., serverless computing. A line of previous work has focused on integrating confidential computing with serverless computing. S-FaaS [20] integrates Intel SGX with OpenWhisk to build a confidential FaaS solution, which can be integrated with smart contracts to enable decentralized payment. Se-Lambda [21] leverages confidential enclaves to protect the API gateway and service runtime in the serverless computing. AccTEE [22] combines confidential computing with Web-Assembly to design a two-way sandbox that offers remote computation with resource accounting trusted by consumers and providers. T-FaaS [23] ports JavaScript engines into encrypted enclave to build a secure serverless platform. In recent years, some works try to optimize the startup latency of confidential serverless computing. Clemmys [24] batches SGX2 `EAUG` operations for fast creation of large-heap serverless enclave. Plug-in Enclaves [2] introduces the plugin enclaves to reuse attested common states, so that function startup latency can be significantly reduced. Reusable Enclaves [25] enable the reusing of SGX enclave by rewinding the WebAssembly runtime, mitigating the cold startup latency.

Optimistically speaking, in the near future, confidential computing will become the new generation of cloud infrastructure and support various application systems concerning data security and user confidentiality.

Part I

New Side-channel Investigation in Confidential Computing

Chapter 3

Automated Extraction of Automated Machine Learning Models

In this chapter, I present `NASPY`, the first end-to-end adversarial framework to extract the network architecture of novel deep learning models automatically generated by Neural Architecture Search (NAS), even when those models are well sealed inside the black-box TEE sandbox. Existing model extraction attacks mainly focus on conventional DNN models with very simple operations, or require heavy manual analysis with lots of prior knowledge. In contrast, `NASPY` introduces seq2seq models to automatically identify novel and complicated operations (e.g., separable convolution, dilated convolution) from hardware side-channel sequences of model inference, allowing it to bypass the TEE isolation and protection. Furthermore, I present methods to recover the model hyper-parameters and topology from the operation sequence. With these techniques, `NASPY` is able to extract the complete NAS model architecture with high fidelity and automation.

3.1 Introduction

Recently Automated Machine Learning (AutoML) has attracted lots of attention from the machine learning community, as it can significantly simplify the development of machine learning pipelines with high efficiency and automation. One of the

most popular AutoML techniques is Neural Architecture Search (NAS) [121, 122], which can automatically generate high-quality Deep Neural Networks (DNNs) for a specified task. It enables non-experts to produce machine learning architectures and models which can outperform hand-designed solutions.

From the adversarial perspective, this chapter aims to design new attacks to *steal* the architectures of black-box NAS models. This is known as *model extraction attacks*, which could cause severe consequences: (1) searching a good architecture with NAS is an energy- and time-consuming process. Hence the produced architecture is naturally considered as an important intellectual property, and stealing it can lead to copyright violation and financial loss [123]. (2) Extracting the model architecture can facilitate other black-box attacks, e.g., data poisoning [124], adversarial examples [125], membership inference [126].

One solution of model extraction is to remotely query the target model and recover the architecture based on the responses [127]. However, such attack requires large computation cost and can only be applied to simple neural networks[1]. Then I turn to a more promising solution: hardware attacks (e.g., cache side-channel, bus snooping). Essentially, when a DNN model executes the inference task on a computer, it leaves architecture-dependent footprints on the low-level hardware components, which could be captured by an adversary to analyze and recover the high-level architecture details. These techniques can give very fine-grained information, and have been utilized by prior works [80, 83, 84, 128] to extract the architectures of conventional DNN models. It is also the most effective method for breaching the isolation of TEE sandboxes. However, there are several challenges when I apply such attack techniques to extract NAS architectures. (1) These works can only handle simple operations in conventional models, while failing to analyze new operations introduced by NAS (e.g., separable convolution, dilated convolution). (2) Some works need complicated manual analysis with prior knowledge of the victim model. For instance, [80] requires the information of the victim model family, and can only extract variants of generic architectures. [128] needs to know the layer type ahead.

To the best of the knowledge, there is only one work [123] focusing on the extraction of NAS models, which is not very practical or general. (1) This work mainly monitors the API traces in the high-level deep learning library, which requires the

[1]It takes 40 GPU-days to recover a 7-layer architecture with a simple chained topology [127].

attacker and victim to share the same library with exactly the same version. This is not practical since users may use different libraries, especially their customized ones. (2) It needs the accurate dimension estimation to predict the layer type and model topology, which is hard to obtain in the real world. (3) The NAS model considered in this work is too small and simple, which cannot well represent state-of-the-art NAS techniques.

I propose `NASPY`, a learning-based framework for automated extraction of NAS architectures with high efficiency and fidelity. I make several contributions to overcome the above limitations. First, I exploit cache side-channel techniques to monitor the low-level BLAS library. Hence, the framework can be applied to different platforms regardless of the high-level deep learning libraries (Tensorflow, Pytorch, or other customized libraries). This cannot be achieved in [123]. Second, I model the extraction attack as a sequence-to-sequence problem, and design new deep learning models to predict the model operation sequence automatically. This does not require the tedious manual analysis, as conducted in [80]. Meanwhile, the models are able to predict new sophisticated operations in NAS, which are missing in [80, 83]. Third, I propose a new analysis method to precisely recover the exact hyper-parameters without any prior knowledge. In contrast, previous works can only estimate a possible range of hyper-parameter values [80, 83]. Finally, I design strategies to reconstruct the model topology and extract the complete architecture for different scenarios and adversarial goals.

I perform extensive experiments to demonstrate the effectiveness of `NASPY`. My identification model can predict the operation sequences of different NAS methods (DARTS [129], GDAS [130] and TE-NAS [131]) with an error rate of 3.2%. My hyper-parameter prediction can achieve more than 98% accuracy. The framework also demonstrates high robustness against random noise introduced by the complex and dynamic hardware systems.

3.2 Background

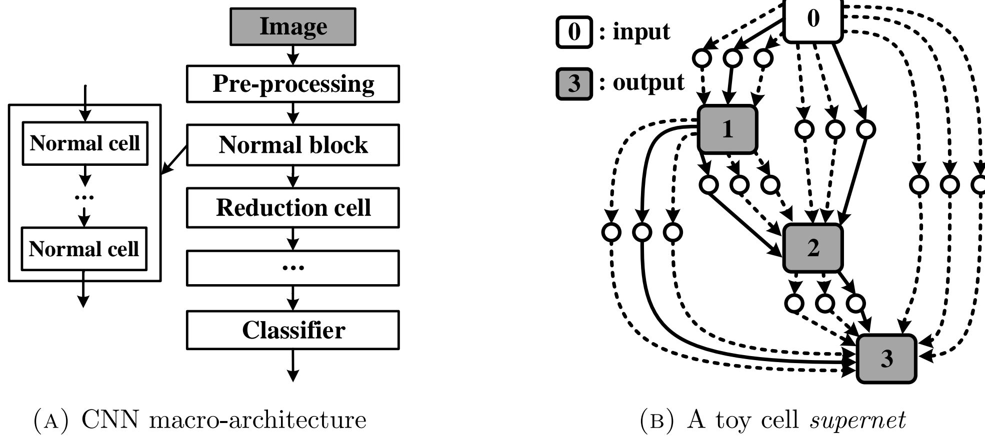
3.2.1 Neural Architecture Search (NAS)

NAS [121, 122] has gained popularity in recent years, due to its capability of building machine learning pipelines with high efficiency and automation. It systematically searches for good network architectures for a given task and dataset. Its effectiveness is mainly determined by two factors:

Search space. This defines the scope of neural networks to be designed and optimized. Instead of searching for the entire network, a practical strategy is to decompose the target neural network into multiple *cells*, and search for the optimal structure of a cell [132]. Then cells with the identified architecture are stacked in predefined ways to construct the final DNN models. Figure 3.1a shows the typical architecture of a CNN model based on the popular NAS-Bench-201 [133]. It has two types of cells: a *normal cell* is used to interpret the features and a *reduction cell* is used to reduce the spatial size. A block is composed of several normal cells, and connected to a reduction cell alternatively to form the model.

A cell is generally represented as a directed acyclic graph (DAG), where each edge is associated with an operation selected from a predefined operation set [134]. Figure 3.1b gives a toy cell *supernet* that contains four computation nodes (squares) and a set of three candidate operations (circles). The solid arrows denote the actual connection edges chosen by the NAS method. Such *supernet* enables the sharing of network parameters and avoids unnecessary repetitive training for selected architectures. This significantly reduces the cost of performance estimation and accelerates the search process, and is widely adopted in recent methods [129, 130, 135, 136].

Search strategy. This defines the approach to seek for good architectures in the search space. Different types of strategies have been designed to enhance the search efficiency and results, based on reinforcement learning [121, 132, 134], evolutionary algorithm [137, 138] or gradient-based optimization [129, 130, 139]. My watermarking scheme is general and independent of the search strategies.



(A) CNN macro-architecture (B) A toy cell *supernet*

FIGURE 3.1: Architecture of a NAS model based on cells

3.2.2 Hardware attacks

Following the previous works [80, 83, 123], I aim to exploit some hardware attacks to perform the extraction of NAS models. Specifically, I consider the following two attacks.

Cache side-channel attacks: CPU caches are introduced between the CPU cores and main memory to accelerate the memory access. Since the attacker can share the same cache with the victim, he can reveal the behavior pattern of the victim from the contention on the usage of cache lines. In this chapter, I adopt FLUSH-RELOAD, which is also used in [80, 123] for model extraction. The adversary leverages the shared low-level BLAS library to infer the victim model, and is able to obtain the sequence of its critical operations, e.g., the matrix multiplication.

Bus snooping attack: data traffic between the processor and memory system is achieved through a *bus*, which sends data to or loads data from specific addresses. Hence, by observing the memory traffic through the bus, the attacker can obtain the memory address traces of the victim model, which further reveals the connections between model layers. In this chapter, I use the bus snooping technique [140] to monitor the read/write addresses of each model layer, which is also adopted in [83]. The adversary can only observe the data addresses and cannot access the data passing through buses, which allows NASPY to work even when the model is encrypted inside a TEE sandbox.

3.2.3 Sequence-to-sequence learning

Seq2seq learning is raising increased attention in the machine learning community, and becoming quite popular for different tasks like speech recognition [141], machine translation [142], image captioning [143], question answering [144], etc. Three models are mostly used for seq2seq learning: Recurrent Neural Network (RNN), Connectionist Temporal Classification (CTC), and attention models (Transformer). In this chapter, I aim to use seq2seq learning for automated model extraction from the monitored memory activities. I design an RNN-CTC model and a Transformer model to recover the structure operations of NAS models.

3.3 Framework Overview

3.3.1 Threat Model

Adversary's goal. Given a sealed victim model M constructed from NAS, the adversary aims to recover a similar network architecture as M, without searching it with large cost and the original dataset. I consider two types of goals following [145]: (1) *accuracy extraction*: the adversary aims to reproduce a network architecture, which can give similar model accuracy as the victim model; (2) *fidelity extraction*: the adversary wishes to recover the same architecture and hyper-parameters as the victim one.

Adversary's capability. I consider two practical scenarios for extracting model architectures. For each scenario, I assume the attacker only knows the target model is constructed by NAS, without any other prior knowledge, e.g., the model family, layer type, NAS method, high-level deep learning library. Figure 3.2 illustrates the overview of the threat model.

- *Remote attack*: model extraction in this scenario is adopted in [80, 123]. The attacker can launch his malicious program on the same machine with the victim model. Although these programs are isolated by the OS or even TEE sandbox, the attacker can still exploit the cache side-channel technique to monitor the victim's low-level executions, e.g., the sequence of matrix multiplication events.

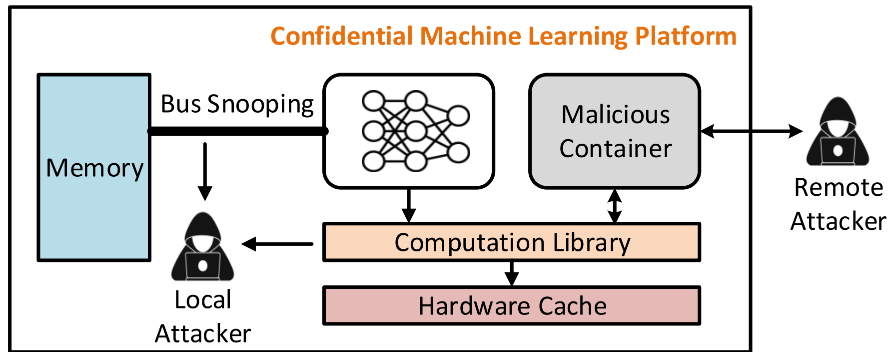


FIGURE 3.2: Overview of threat model.

Based on the side-channel sequence, the attacker is able to perform accuracy extraction of the victim model.

- *Local attack*: this scenario is considered in [83], where the attacker can physically access the machine running the victim model. In addition to launching the cache side-channel attack to retrieve the operation sequence, the attacker can also launch the bus snooping attack to monitor the memory bus and PCIe events. With such memory address traces, the attacker can achieve fidelity extraction of the victim model and recover the exact model topology.

3.3.2 Attack Overview

Before describing my NASPY framework, I need to understand the workflow of the model inference process. As shown in the left diagram of Figure 3.3, given a DNN model, a deep learning library (e.g., Tensorflow, Pytorch) is used to process the computational graph of the model, and convert the model architecture into sequences of connected layer operations. Then these operation sequences are sent to low-level computation libraries for acceleration, such as the BLAS library (e.g., *OpenBLAS*) for GEneral Matrix Multiplication (GEMM), and the mathematical library (e.g., *libm*) for activation functions. Those computations will be executed on the hardware platform. By monitoring the hardware activities using cache side-channel and bus snooping techniques, the attacker can observe the event sequences and memory address traces for the model inference process.

Overview of NASPY. The adversary's task is to automatically and precisely recover the NAS model architecture from the captured sequences of hardware activities.

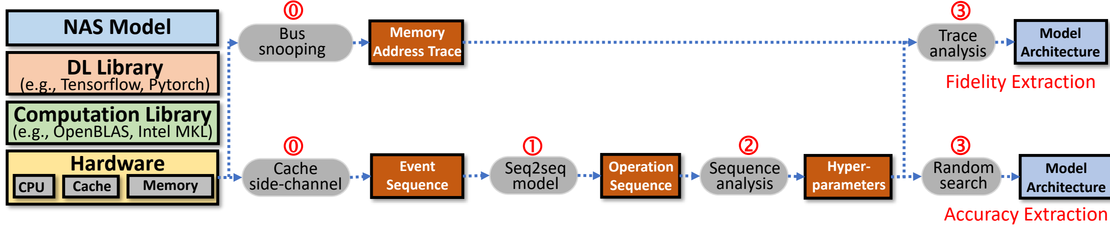


FIGURE 3.3: Workflow of my model extraction framework.

Figure 3.3 shows the workflow of my NASPY framework. It consists of three steps, as described below.

First, it translates the event sequences from cache side-channel attacks to the operation sequences (i.e., input of the low-level computation library). I model this as a seq2seq problem, and design two deep learning models (RNN-CTC and Transformer) to achieve this prediction. Second, it identifies the values of hyper-parameters in the layer operations recovered from the first step. Previous works [83] can only estimate a range of these values based on the dimension size of layer input and output. In contrast, NASPY can precisely reveal the exact values of the hyper-parameters from the translated operation sequence. Third, NASPY reconstructs the model topology and obtains the complete architecture. For the *accuracy extraction* attack, the attacker can randomly select a model topology, and assign the recovered operations with the hyper-parameters to it. My experiment results show that the corresponding model can give similar accuracy as the victim one. For the *fidelity extraction* attack, the attacker needs to recover the exact model topology. He can leverage the information in the memory address trace from the bus snooping attack to construct the architecture.

3.4 Detailed Design

3.4.1 Operation Sequence Identification

The first step is to predict the operation sequence from the hardware event sequence. Past works [80, 123] require manual analysis for such translation. In contrast, I propose to leverage a seq2seq deep learning model to achieve this task automatically. Before discussing my method, I first give a short introduction about the attack target, the OpenBLAS library.

Details about GEMM in OpenBLAS. BLAS realizes the matrix multiplication with the function *gemm*. This function computes $C = \alpha A \times B + \beta C$, where A is an $m \times k$ matrix, B is a $k \times n$ matrix, C is an $m \times n$ matrix, and both α and β are scalars. OpenBLAS adopts Goto's algorithm [146] to accelerate the multiplication using modern cache hierarchies. This algorithm divides a matrix into small blocks (with constant parameters P, Q, R), as shown in Figure 3.4. The matrix A is partitioned into $P \times Q$ blocks and B is partitioned into $Q \times R$ blocks, which can be fit into the L2 and L3 caches, respectively. The multiplication of such two blocks generates a $P \times R$ block in the matrix C. Algorithm 1 shows the process of *gemm* that contains 4 loops controlled by the matrix size (m, n, k). Functions *itcopy* and *oncopy* are used to allocate data and functions. *kernel* runs the actual computation. Note that the partition of m contains two loops, $loop_3$ and $loop_4$, where $loop_4$ is used to process the multiplication of the first $P \times Q$ block and the chosen $Q \times R$ block. For different cache sizes, OpenBLAS selects different values of P, Q and R to achieve the optimal performance.

Algorithm 1 GEMM

Input: matrice A, B, C; scalars α, β
Output: $C = \alpha A \times B + \beta C$

```
for j in (0:R:n) do // Loop 1
    for l in (0:Q:k) do // Loop 2
        call itcopy;
        for jj in (j:3UNROLL:j+R) do // Loop 4
            call oncopy; call kernel
        for i in (P:P:m) do // Loop 3
            call itcopy; call kernel
```

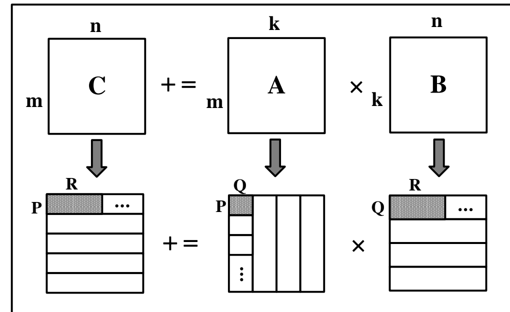


FIGURE 3.4: GEMM procedure.

Dataset formulation and preprocessing. With knowing the internal mechanism of GEMM implementation, I can start to conduct the dataset for training my analysis model. Specifically, the event sequence x is a time-series of length T, where each frame is a vector of event features. In this chapter, I capture the occurrence of *itcopy* and *oncopy* APIs in OpenBLAS, which are used to load matrix data for GEMM. Note that my framework can be simply generalized to other BLAS libraries, such as Intel MKL. Hence, a frame is denoted as $x_i = (I_i, O_i, T_i)$, where T_i is the time interval from the last monitoring moment, and I_i and O_i are binary values to denote whether *itcopy* and *oncopy* are called during this interval. T_i is determined by the monitoring granularity when collecting side-channel information.

The operation sequence y contains N operations performed by the victim model. To comprehensively cover novel neural architectures, I consider all the common operations used in state-of-the-art NAS methodologies: *fully connected layer (FC)*, *normal convolution (Conv)*, *dilated convolution (DConv)*, *separable convolution (SConv)*, *dilated-separable convolution (DSConv)*, *pooling (Pool)*, *identity (Skip)*, *zeroize*. It is easy to integrate other operations into `NASPY` if necessary.

Figure 3.5 shows the event sequences of four representative operations, where the blue and red nodes denote the occurrences of *itcopy* and *oncopy*. I observe that different operations have distinct event patterns, giving us opportunities to predict the model operations from the side-channel leakage. However, there are a couple of challenges to design seq2seq models for this task. I perform the following data preprocessing methods to overcome these challenges.

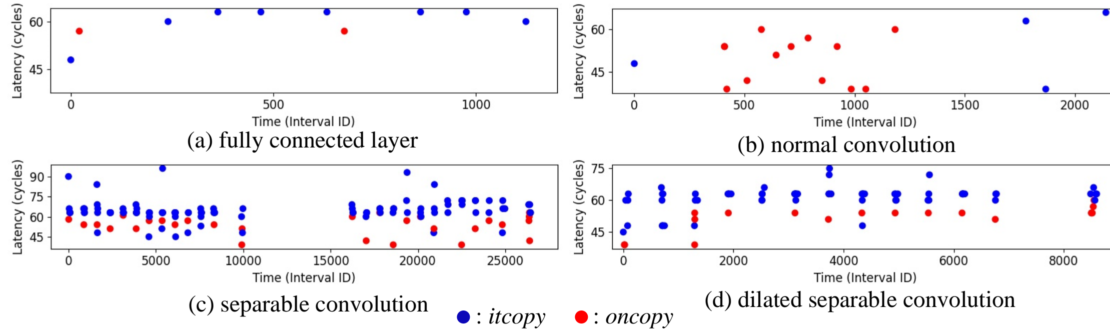


FIGURE 3.5: Event sequences of four representative operations in NAS models.

1. *Input downsampling and label extending.* An event sequence x of a NAS model is extremely long (> 5 million frames) because of the high monitoring frequency. In contrast, an operation sequence y usually has only hundreds of operations. It is infeasible to directly use existing seq2seq learning models to handle such length gap. To resolve this challenge, I perform downsampling for x and extend labels in y. Specifically, for the event sequence x, I only keep the frames whose I_i or O_i is 1. I also update T_i in these active frames as the time interval from the last function access. This can significantly reduce the sequence length, while preserving the critical information. For the operation sequence y, I decompose the single label of a complex operation into a series of sub-operation labels. For instance, a *SConv* operation is composed of several *Conv* sub-operations (Figure 3.5(c)). So instead of directly labeling this operation as *SConv*, I extend it as multiple *Conv* labels, the number of which is determined by the channel size. Such preprocessing can remarkably decrease the difficulty of training the seq2seq model.

2. *Inter-operation context.* Not every operation in y has the corresponding event in x. For instance, *Pool* and *Skip* never invoke GEMM computations. This makes it difficult to predict such operations. I propose to leverage the execution latency and inter-operation context for identification, since their values are different in these operations. Specifically, I introduce two interval labels into the operation sequence: I_Γ denotes the interval between two operations while I_γ denotes the interval between the decomposed sub-operations within an operation. Such two labels reflect the context switch for inter- and inner-operations, which hence helps the seq2seq model distinguish adjacent operations in the sequence with high accuracy. Experiment results in Section 3.5.1 show that the consideration of inter-operation context brings large performance improvement.

3. *Time normalization.* When the victim model runs on different platforms, although the occurred events (i.e., I_i and O_i) keep constant, the time interval T_i between the frames will be different, due to the varied execution latency in hardware. This can restrict the generalization of my seq2seq model. To overcome this challenge, I perform normalization over T_i in the captured side-channel trace, and use the relative intervals to train the model, which are similar across various platforms as they are determined by the algorithm behaviors.

4. *Data augmentation.* Different from NLP and CV tasks, the raw side-channel data can contain lots of random noise from the hardware activities. This is due to the complex system optimization and run-time dynamics [83]. Such noise can decrease the identification accuracy. To improve the robustness of the seq2seq model against noise, I further perform data augmentation following [147], which simply cuts out random blocks of consecutive time and feature dimensions. In this chapter, I just mask these blocks with the fixed value 0.

Seq2seq model designs. I propose two kinds of models for predicting the operations from the side-channel trace. The first one is an **RNN-CTC model**. Recently the combination of RNN and CTC decoders is commonly used in sequence modeling problems. Figure 3.6(a) shows the architecture of this model for identifying the operation sequence y from the event sequence x. Given that x only contains three features in each frame, I first introduce a convolution layer to learn more features. Then an RNN layer is used to propagate information through this sequence. I adopt the Bidirectional Gated Recurrent Unit (BGRU) as the RNN module, which can enable better long-term memory and fully leverage the past and future

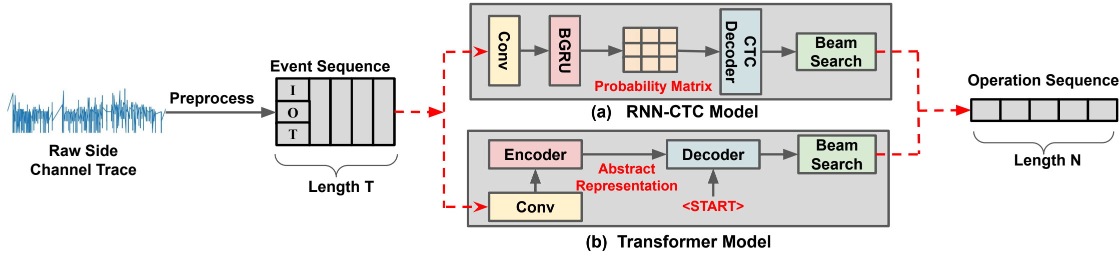


FIGURE 3.6: Procedure of operation sequence identification.

contexts. The output of the RNN layer is the probability distribution of all the operation labels for each frame, which is further fed into the CTC decoder. While it is difficult to align the operation sequence with the event sequence that has a various length, the CTC decoder can skip the alignment by introducing a "blank" label. Finally, the operation sequence y with the largest prediction probability is identified with the beam search.

The second one is a **transformer model.** Proposed by [148], transformers have broken multiple AI task records and pushed the state of the art. Hence, I also show the possibility of operation identification with a transformer model, as shown in Figure 3.6(b). Transformer is an attention-based neural network with a typical encoder-decoder architecture. The encoder maps the input sequence into an abstract representation that contains all the learned features of that input. Then the decoder takes this abstract representation and predicts the next output step-by-step based on the previous output. The introduction of the attention mechanism enables the transformers to have extremely long term memory, which can focus on all the tokens generated previously. Similarly, I add a convolution layer before the encoder to learn more features from the event sequence.

3.4.2 Hyper-parameter Recovery

My second step is to extract the architectural hyper-parameter values of each operation. NASPY can precisely recover the exact hyper-parameters, rather than an estimated range in previous works [80, 83].

Convolutions. Given that convolutions take a majority in a NAS model, I first discuss how to reveal hyper-parameters of different convolution operations, e.g., kernel size R, dilation d, channel size C, padding size P and stride. Note that I assume the input size (i.e., width and height) has been identified from the analysis

of the previous layer, while the initial input size of the model is publicly available. (1) The extended event labels in y is integrated to compose the single label of the complex operation (e.g., *SConv* and *DSConv*). During this process, NASPY discovers the channel size C by counting the number of event labels. (2) Both the kernel size R and dilation d can be directly recovered from the predicted operation labels (e.g., 3×3 *SConv* or 5×5 *DSConv*). NASPY can distinguish various convolutions with different kernel sizes, as they have different side-channel patterns and execution time. (3) Empirically, the padding size P is normally set as $R/2$ to keep the input size and output size constant for possible residual connections. (4) The stride can be deduced from the cell type. Normal cells keep the spatial size unchanged while reduction cells would half the spatial size but double the channel size to maintain the dimension information. Hence, the stride is 1 for normal cells and 2 for reduction cells. The cell type is identified through the channel size C.

Other operations. I can also infer the hyper-parameters for other operations like the *FC* and *Pool* layers. The number of neurons in a *FC* layer can be reflected by the length of the event sequence, where a larger number of neurons leads to a longer sequence. Such length variance can be well learned by the seq2seq model, which discloses the number of neurons in the predicted labels. For the *Pool* layer, the relationship between the pooling size R and padding size P is $P = R/2$ in order to keep the spatial size among layers in the same cell.

3.4.3 Model Topology Reconstruction

The final task is to extract the topology to obtain the model architecture. As a NAS model is built with cells, I also reconstruct the model topology in term of cells. First, the model macro skeleton is determined by analyzing the number and types of cells from the event sequence. It is intuitive to locate each cell, since they are separated by much larger time intervals due to some extra computations like concatenating and preprocessing. Then, I focus on the topology of each cell. Unlike conventional DNN models, the topology in NAS cells is not chained and sometimes even not regular. Hence, I cannot simply connect each extracted operation in the sequence to form the topology. Based on the attack scenarios and goals, the cell reconstruction can be done with different methods.

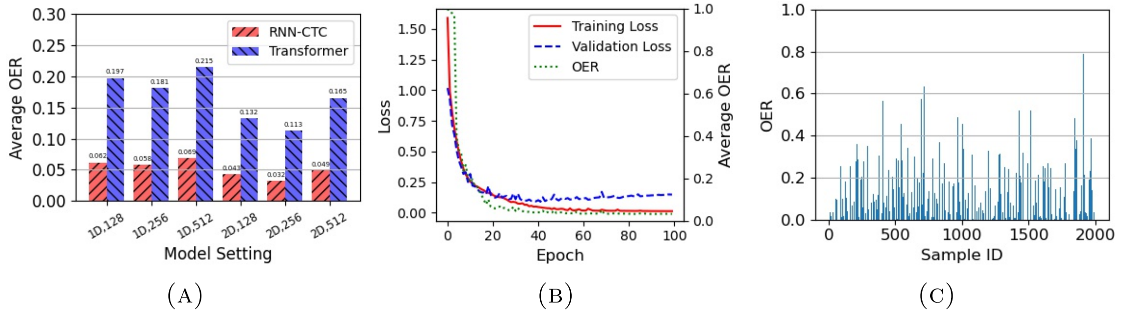
Accuracy extraction. To extract a model architecture with similar accuracy, the attacker can just use the remote side-channel attack to recover the operations and hyper-parameters, and then randomly choose a topology following the basic rule of NAS, i.e., each neuron node has two inputs. The experiment results in Section 3.5.3 verify the effectiveness of this strategy. Previous works [129, 130] have also noted that operations have a greater impact on NAS model performance than topology.

Fidelity extraction. As only the side-channel event sequence is not enough to achieve fidelity extraction, the attacker can adopt bus snooping to get the memory address trace to reveal the exact interconnections between layers. With the revealed model topology, the attacker can finally extract the complete model architecture.

3.5 Evaluation

Dataset construction. I search model architectures with CIFAR10, and train model parameters over CIFAR10 and CIFAR100. My method can be applied to tasks for other datasets as well. I first generate 10,000 random computational graphs of NAS models following the macro skeleton proposed in the popular benchmark NAS-Bench-201 [133], where each cell contains 4 nodes associated with 8 operations (each node has two inputs). The operation set follows the classical set in [129], which contains: *identity*, *zeroize*, *3×3* and *5×5 SConv*, *3×3* and *5×5 DSConv*, *3×3 average pooling*, *3×3 max pooling*. Conventional operations (i.e., *Conv* and *FC*) are also included in my model.

Then I collect the corresponding side channel sequences using the Flush-Reload technique, which inspects the cache lines storing OpenBLAS functions (*itcopy* and *oncopy*) at a granularity of 2000 CPU cycles. I randomly select 80% of the sequences as the training set, and the rest as the validation set. The seq2seq models will be tested on novel NAS models generated by three state-of-the-art NAS methods: DARTS [129], GDAS [130] and TE-NAS [131]. For each method, I search 10 NAS models with various initial seeds for testing.



(A) (B) (C)

FIGURE 3.7: (a) Average OER of the two identification models with different configurations. (b) Loss and OER trend of the RNN-CTC model. (c) OER of the transformer model on validation samples

3.5.1 Operation Sequence Identification

Metrics. Inspired by the evaluation metric WER (Word Error Rate) in NLP tasks, I propose Operation Error Rate to quantify the prediction accuracy. It is calculated as: $OER = D(y', y)/|y|$, where $D(y', y)$ is the edit distance between the predicted operation sequence y' and ground-truth sequence y, and $|y|$ is the sequence length of y. A smaller OER implies higher identification accuracy.

Prediction accuracy. Given that my dataset is relatively small, I only use one layer of BGRU in the RNN-CTC model, and one layer of encoder and decoder in the transformer model. Figure 3.7(a) shows the average validation OER of two models with different structure settings. Both *1D Conv* and *2D Conv* are considered to extract more features from the input, and the model dimension size (i.e., d_{rnn} of the BGRU layer and d_{model} of the transformer) is chosen among [128, 256, 512]. I see that while all the configurations can achieve very high accuracy, the best one is: *2D Conv* and $d_{model} = 256$, where both models give the lowest OER: 3.2% for RNN-CTC and 11.3% for the transformer. The trend of the loss and average OER during the training of RNN-CTC are depicted in Figure 3.7(b). I observe that the loss and OER decrease dramatically within the first 20 epochs and reach convergence at epoch 40. In contrast, the transformer model performs relatively worse. Figure 3.7(c) shows the OER of each validation sample predicted by the transformer, which is higher and more unstable. It is because the length of the preprocessed side-channel event sequence (normally 5000 frames) is still too long for the transformer, which requires tremendous GPU and memory resources, making the training more difficult.

Model	Average	DARTS	GDAS	TE-NAS
RNN-CTC	0.032	0.035	0.021	0.038
Transformer	0.113	0.151	0.115	0.094

TABLE 3.1: Testing OER for three NAS methods.

Finally, I test the above two models on the NAS models generated by DARTS, GDAS and TE-NAS[2], and the results are shown in Table 3.1. From the table, both of the RNN-CTC and transformer models can well predict the operation sequence from the side-channel leakage of three types of NAS models.

Effectiveness of inter-operation context. I adopt the *inter-operation context* technique to handle the missing events of some operations (Section 3.4.1). To evaluate its effectiveness, Figure 3.8 compares the prediction error rates of the two models with and without considering the context. Figure 3.8(a) shows the OER trend of the RNN-CTC model during training. I observe that without the inter-operation context, the OER decreases more slowly and converges at a higher value. Figure 3.8(b) depicts the fitting curves on the OER of validation samples for the transformer. Ignoring the context will lead to a huge performance penalty, which gives higher and less stable prediction error rates.

Robustness against noise. I further evaluate the robustness of my models against the noise. Figure 3.9 shows the OER of two models without and with the data augmentation technique, when the side-channel event trace contains different scales of random noise. First, I observe that my two models have strong robustness and give acceptable error rates under large amounts of noise. Second, data augmentation (masking rate = 0.1) can further improve the model robustness. With 30% random noise, the OER of RNN-CTC is 0.115 and the transformer is 0.208. Besides, data augmentation has better improvement for RNN-CTC than the transformer, as the transformer always considers the entire input, which weakens the effect of this technique.

[2] I train these models using the open-sourced code from the authors.

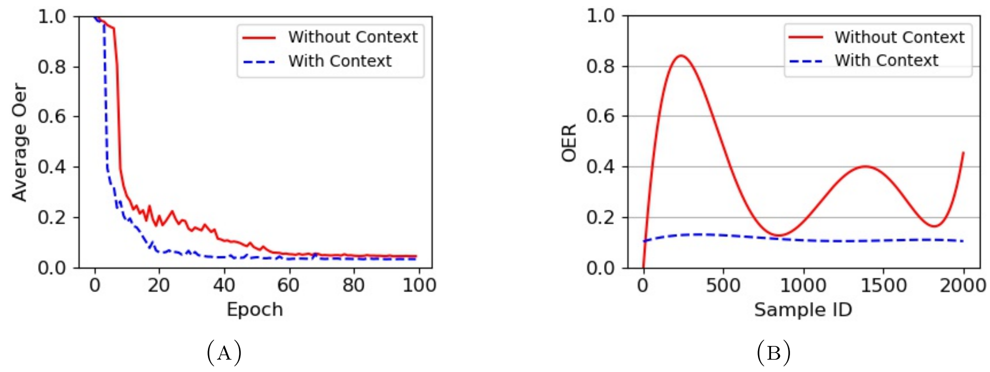


FIGURE 3.8: Inter-operation context testing. (a) OER trend of RNN-CTC. (2) OER of the transformer on validation samples.

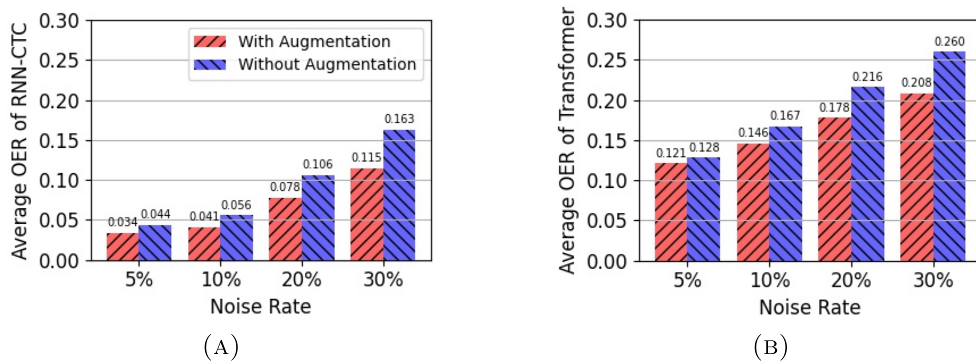


FIGURE 3.9: Robustness versus different scales of noise. (a) OER of RNN-CTC. (b) OER of the transformer.

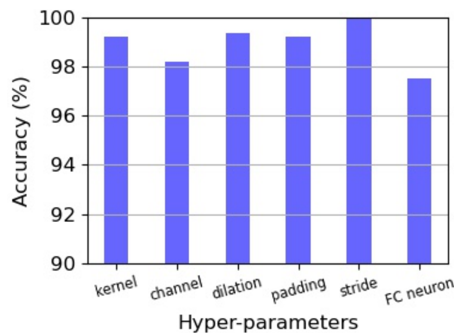


FIGURE 3.10: Recovery accuracy.

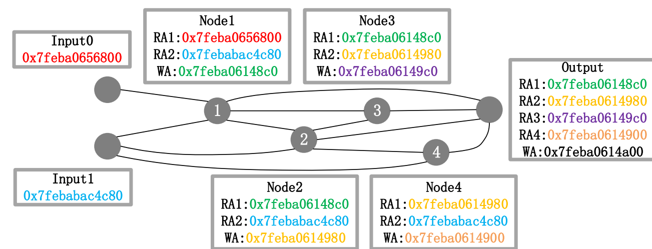


FIGURE 3.11: Memory address trace of a NAS cell.

3.5.2 Hyper-parameter Recovery

NASPY can further extract the hyper-parameter values from the predicted operation sequence. As RNN-CTC performs better, I conduct hyper-parameter recovery based on the operation sequence from this model. The hyper-parameter prediction

error is identical to the operation identification error (3.2%). Figure 3.10 shows the recovery accuracy of various hyper-parameters. For convolutions, the accuracy of each hyper-parameter can reach up to 98%. The kernel size and dilation are recovered directly from the predicted labels and have the highest accuracy. The padding size is computed from the kernel size with the same accuracy. The stride is inferred from the cell type, which can reach 100% accuracy, as normal and reduction cells are very distinguishable in terms of the channel size. The recovery accuracy of the channel size is relatively lower, because most errors in the predicted operation sequence are from the adding or deletion of events, while the channel size is revealed by counting the repetitions of events in a convolution. However, such slight accuracy drop can be easily compensated with post-analysis. For the number of neurons in FC layers, the recovery accuracy is 97.54%, as the side channel pattern of FC is shorter and simpler, making it hard to be distinguished sometimes. The pooling size cannot be directly reveal from the operation sequence, as the changing of pooling size does not lead to significant changes of the side-channel leakage pattern. So for this specific operation, I can determine its value empirically from the common values (i.e., 3 or 5).

3.5.3 Model Topology Reconstruction

First, I consider the accuracy extraction attack. After the attacker identifies the operations and hyper-parameters, he can randomly choose a model topology to obtain the architecture, which can give close model performance. To validate this, given a victim NAS model, I first extract its operation sequence and hyper-parameters. Then I randomly generate a computation graph that connects nodes in the cell, and associate the revealed operations to the graph sequentially. I train the model from this graph and test its prediction accuracy. Table 3.2 shows the results on CIFAR10 and CIFAR100 datasets with 5 different graphs. I can see that the randomly chosen topology can give very similar accuracy as the original model, where the accuracy drop is less than 1% for both datasets. I also check the performance of totally randomly constructed models

Dataset	Original Model	Random Model with Same Operations				
		#1	#2	#3	#4	#5
CIFAR 10	96.82	96.53	96.44	96.60	96.57	96.77
CIFAR 100	81.07	80.95	80.16	80.33	80.90	80.56

TABLE 3.2: Accuracy (%) of random models on two datasets.

(operations and topology): the average accuracy of 5 such models is 91.85% (CIFAR10) and 72.49% (CIFAR100), which is much lower than my extracted models. This shows the importance of operation and hyper-parameter recovery for accuracy extraction.

Second, I consider fidelity extraction. The memory address trace of the victim model can be obtained with bus snooping tools, e.g., HMTT-v4 [140]. Figure 3.11 shows an example of the monitored address trace of a cell in the victim model, where `RA` is the *read* address and `WA` is the *write* address. If `RA` of one layer a is the same as the `WA` of another layer b, I confirm that there is a connection from b to a. By analyzing the consistency between `RA` and `WA` in the address traces, the connections between layers can be recovered precisely. I perform experiments on 6 NAS models in Table 3.2 and the results confirm that the model topology can be revealed with 100% precision. Combining the recovered operations and hyper-parameters, which may contain some small errors, I can finally reconstruct the complete architecture that is almost the same as the original one.

3.6 Discussions

Extracting other NAS models. My framework can be extended to attack other types of NAS models as well. For instance, some latest NAS techniques (e.g., [149]) do not adopt the cell-based structure for searching. Since `NASPY` focuses on the recovery of operation sequences rather than cells, it is still effective to extract models from those NAS solutions. Another example is NAS-based RNN models. A RNN model generated by a NAS method from the search space only contains activation functions [129]. It is noted that these functions cannot be observed from the GEMM trace. Instead, I can monitor other libraries (e.g., libm) to identify the functions. In the future, I will design seq2seq models and methods for extracting these models.

Extracting standard DNN models. My framework can also be generalized to conventional standard DNN models. For models with chained topology (e.g., VGG), it is easy to extract their model architectures by first predicting the operation sequence and then just connecting them in sequence. However, for models with more complex topology (e.g., ResNet), after revealing the operation sequence,

I need to perform fidelity extraction to reveal the exact model topology. If I know the family of the target model (assumed in [80]), I can also reconstruct a similar model architecture by connecting the predicted operations as the template structure of the model family.

Defense strategies. There are several solutions that can possibly mitigate my extraction attacks. From the hardware perspective, oblivious RAM [150] was designed to hide the memory access pattern and thwart bus snooping attacks. Latest TEE designs [6] like Intel TDX also tries to hide memory access trace from potential attackers. Security-aware cache architectures [151, 152] were proposed to reduce side-channel leakage. From the application perspective, I can obfuscate the model's execution behaviors by adding dummy operations or shuffling the operation orders [123]. However, those solutions either require significant changes to the hardware, or add large computation overhead to the model inference. I will systematically evaluate those defenses, and design more efficient approaches as future work.

3.7 Conclusion

In this chapter, I design `NASPY`, an end-to-end framework to breach the protection of TEE sandbox and achieve automated extraction of novel NAS model architectures. `NASPY` adopts new deep learning models to identify model operation sequences from the side-channel event trace, which allows the attacker to bypass the TEE isolation. With the identified operations, `NASPY` can further precisely recover the operation hyper-parameters and model topology for various scenarios. Compared to past works, `NASPY` can extract novel operations and architectures with higher automation and accuracy, bringing more severe threats to the protection of AI systems, even if they are protected with confidential computing infrastructures. I expect this study can raise the awareness of the community about the severity of model extraction attacks and the vulnerability in existing confidential AI systems, inspiring researchers to come up with more secure solutions to protect the architecture privacy.

Chapter 4

Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels

In modern cloud platforms, it is becoming more important to preserve the privacy of guest virtual machines (VMs) from the untrusted host. To this end, Secure Encrypted Virtualization (SEV) is developed as a hardware extension to protect VMs by encrypting their memory pages and register states. Unfortunately, such confidential VMs are still vulnerable to micro-architectural side channels, and Hardware Performance Counters (HPCs) are a prominent information leakage source. To make matters worse, currently there is no systematic defense against the HPC side channels. I introduce `Aegis`, a unified framework for demystifying the inherent relations between the instruction execution and HPC event statistics, and defending VMs against HPC side channels with provable privacy guarantee and minimal performance overhead. `Aegis` consists of three modules. Application Profiler profiles the application offline and adopts *information theory* to quantitatively estimate the vulnerability of HPC events. Event Fuzzer leverages the *fuzzing* technique to automatically generate interesting inputs, i.e., instruction sequences, that can effectively alter the HPC observations. Event Obfuscator injects noisy instructions into the protected VM based on the *differential privacy* mechanisms for high efficiency and privacy. I present three case studies to demonstrate that `Aegis`

can defeat different types of HPC side-channel attacks (i.e., website fingerprinting, DNN model extraction, keystroke sniffing). Evaluations show that `Aegis` can effectively decrease the attack accuracy from 90% to 2%, with only 3% overhead on the application execution time and 7% overhead on the CPU usage.

4.1 Introduction

The maturity of cloud computing ecosystems prompts an increased emphasis on the privacy guarantee, making it a top concern along with the performance efficiency for cloud service providers and customers. To protect the guest virtual machine (VM) from the privileged but potentially malicious hypervisor, a promising mechanism called Trusted Execution Environment [153, 154] is utilized to realize confidential computing in virtualization scenarios. This mechanism adopts a hardware memory encryption engine to encrypt the VM's memory space transparently with a VM-specific key stored in the hardware layer. Even the hypervisor cannot access the encryption key and extract the memory content of the VM. AMD Secure Encrypted Virtualization (SEV) [5] is the first commercial realization of such a mechanism, and has been widely applied in modern cloud services [155, 156]. Other processor vendors have also released similar extensions, e.g., Intel TDX [6] and ARM CCA [157].

Encrypted virtualization can defeat most attacks that directly break the confidentiality or integrity of VM data. However, it is generally vulnerable to side-channel attacks, which use side-channel information (e.g., execution time, memory footprints) to infer the secrets in the encrypted environment. Although the latest SEV-SNP version has provided multiple protections [158] (e.g., Branch Target Buffer Isolation, disabling Instruction Based Sampling) against certain transient-execution and side-channel attacks, AMD admits that "*it is not able to protect against all possible side-channel attacks*" [159]. Over the years, a variety of attacks have been proposed to breach the confidentiality of SEV systems, which establish side channels via performance counters [87], cache occupancy [77], unprotected I/O operations [88, 89], page faults [90–92], etc.

Among these attacks, Hardware Performance Counter (HPC) side channels are particularly easy to exploit. HPCs are implemented as a tool for software profiling, debugging and system modeling. Security researchers also repurposed them for malware and intrusion detection [160–162]. Unfortunately, their capability of inspecting program's behaviors can be abused by the adversary to infer the secret from the victim program [163–167], especially in the cloud scenario where the malicious host can arbitrarily read the HPC register values mapping to a victim VM. While SEV claims HPC leakage can only reveal trivial information about the application inside the VM and hence does not prevent it from the hardware [159], its competitor Intel TDX seriously considers HPC leakage as a target that must be prevented and proposes specific hardware modifications to isolate virtualized guest HPC registers from the malicious host [168]. I present three case studies in Section 4.3 to show that HPC side channels can indeed lead to more severe confidentiality attacks, e.g., leaking the website access, keystroke, and machine learning models. Hence, the confidential VM should well mitigate HPC leakage to protect guest secrets, just as Intel TDX has done.

However, currently all public cloud platforms mainly adopt SEV to achieve their confidential VM services, including Azure confidential VM [169], Google cloud [170] and AWS EC2 [171], while TDX is only a preview version or even has not been considered [172, 173]. Given AMD has announced that SEV-SNP is the latest stable version, it seems that AMD has no motivation to update the hardware design to prevent HPC side channels, which exposes almost all existing confidential VM systems to the security threat. As a result, the lack of hardware defence against HPC side channels in SEV is an urgent and practical security issue, which motivates us to design a practical software-based solution to mitigate this vulnerability for off-the-shelf platforms immediately.

Past efforts have been devoted to reinforce the encrypted VMs and enclaves against different micro-architectural side channels, e.g., CPU cache [7, 8], page faults [9, 10], branch prediction [11, 12]. However, to the best of the knowledge, there are no previous works to systematically and comprehensively investigate the HPC side-channel defenses against malicious hypervisor. This motivates us to design an effective and efficient defense solution for the customers to protect their sealed application from HPC side channels. However, achieving this goal is not a trivial work and several challenges must be carefully addressed. First, modern processors

usually support a large number of HPC events (usually in the thousands). This gives the adversary more flexibility to establish the side channel. Meanwhile, it also gives the defender more difficulties to analyze the possible threats. Second, HPCs cannot count performance events precisely because of the external interference, e.g., hardware interrupts, which may mislead the identification of HPC value changes and result in false analysis results. Finally, one common side-channel defense strategy is to obfuscate the adversary's observations. A straightforward way is to inject random noise directly. However, this method introduces extra noise into VMs, which incurs additional performance overhead. Besides, randomly injecting noise cannot provide a rigorous privacy guarantee.

Motivated by these challenges, I propose `Aegis`, a framework that can automatically analyze the potential HPC side channels of a victim application from customers, and effectively prevent HPC side-channel attacks with a provable security guarantee. `Aegis` consists of three modules:

(1) **Application Profiler**. It is used to profile the protected application and extract all vulnerable HPC events that can act as attack surfaces. It gives us the first look at the application and a comprehensive understanding on usable attack surfaces. The vulnerability of different events are estimated and ranked with *information theory*.

(2) **Event Fuzzer**. It is used to find out all possible instruction gadgets that can alter the profiled vulnerable HPC events. I model this as a bug identification task, and design an automatic tool based on the *fuzzing-like* technique to generate interesting "inputs", i.e., instruction gadgets, to evaluate if the system reports a "bug", i.e., value change in the target HPC events.

(3) **Event Obfuscator**. It is located inside the victim VM and injects specific numbers of instruction gadgets into the execution flow of the VM as the noise to obfuscate HPC values monitored from the outside attacker. The key insight is to model the execution behaviors of the victim VM as statistical data, and then inject random noise following a specific *differential privacy* mechanism. This reduces potential information leakage and makes the victim's activities indistinguishable from the attacker's observations. Meanwhile, the differential privacy mechanism can theoretically guide us to identify the optimal amount of noise, achieving desired privacy guarantee with the minimal impact on the system.

My framework `Aegis` is the first work aiming to systematically and comprehensively mitigate HPC side channels on encrypted VMs. It is a general and provable solution that can be readily deployed inside the VM for a strong privacy guarantee. Two differential privacy mechanisms (Laplace and d^*) are adopted to defeat attacks. Experimental results show that both mechanisms can effectively reduce the attack accuracy from $> 90\%$ to 2%, closing to random guess, while only introducing about 3% overhead on the application execution time and 7% overhead on the CPU usage.

4.2 Background and Related Works

4.2.1 Hardware Performance Counters

Modern processors implement a large spectrum of Hardware Performance Counters (HPCs) in each CPU core to record the occurrences of hardware-related events for processes and the entire computer system. These counters provide developers a useful tool for dynamic software profiling, debugging and system modeling. However, HPCs also expose an exploitable surface, where the attacker can obtain the execution behaviors of a protected program in the system and further recover sensitive information. Prior works showed the feasibility of revealing secret keys from the cryptographic applications based on HPCs [166, 167]. Furthermore, HPCs can also provide high-resolution timing information to facilitate cache attacks [163, 164] or even reveal the website accesses [165].

4.2.2 Secure Encrypted Virtualization

Secure Encrypted Virtualization (SEV) is a new feature in AMD processors [5], which combines AMD-Virtualization (AMD-V) and Secure Memory Encryption (SME) technologies to encrypt individual VMs with their own keys, aiming to protect VMs from the untrusted hypervisor. A dedicated co-processor, Platform Security Processor (PSP), is introduced to generate distinct ephemeral keys for each VM, and encrypts VM's data outside the processor. This extension can effectively protect the secrets in VMs from physical attacks (e.g., cold boot and DMA attacks) and privileged software attacks, making the encrypted VM a total black-box to

the malicious hypervisor. A later SEV-ES [174] version further encrypts VM's CPU register state during world switches, which prevents the malicious hypervisor directly accessing or modifying VM states. The latest SEV-SNP [159] version finally achieves integrity protection of VM memory with Reverse Map Table and fixes vulnerabilities in two earlier versions. Other processor vendors, e.g., Intel [6] and ARM [157], are also working on these security features, which is a promising direction for trustworthy cloud computing.

4.2.3 Fuzzing

Fuzzing is a popular software testing technique to automatically find bugs in software applications [175]. A fuzzer typically generates a significant number of test inputs and monitors software execution over these inputs to detect abnormal behaviors. There are two common fuzzing strategies: (1) mutation-based fuzzers [176, 177] select an initial set of inputs as the seed, and then generate inputs by applying mutations, e.g., splicing, bit flipping. They usually require the analyst to have prior domain knowledge. (2) Grammar-based fuzzers [178, 179] exploit existing input specifications to generate a grammar model to conduct inputs. They sometimes fail to reach certain *corners* of the input space. Recently, hardware fuzzing has become increasingly popular. Researchers adopt this technique to find undocumented x86 instructions [180], discover side channels [181], improve Meltdown [182] and Spectre attacks[183]. Different from those works that use fuzzing to facilitate hardware attacks, I apply it to defeat side channels.

4.2.4 Differential Privacy

This technology was originally used to protect statistical databases by withholding information about individuals [184]. One popular Differential Privacy (DP) solution is ϵ-DP. A randomized algorithm $\mathcal{A} : \mathcal{X} \rightarrow \mathcal{Z}$ satisfies ϵ-DP if for any adjacent datasets x, x' and all $Z \subseteq \mathcal{Z}$, it has: $\mathbb{P}(\mathcal{A}(x) \in Z) \leq exp(\epsilon) \times \mathbb{P}(\mathcal{A}(x') \in Z)$, where $\epsilon \geq 0$ indicates the privacy budget to control the level of privacy protection. Chatzikokolakis et al. [185] proposed a generalization of differential privacy called d-privacy. Compared to ϵ-DP, d-privacy is more suitable for datasets with time series correlations, giving better privacy guarantees under the same privacy budget.

Specifically, a metric d on a set $\mathcal{X}$ is defined as a function $d : \mathcal{X}^2 \rightarrow [0, \infty)$. A randomized algorithm $\mathcal{A} : \mathcal{X} \rightarrow \mathcal{Z}$ satisfies (d, ϵ)-privacy if for all $Z \subseteq \mathcal{Z}$, it has: $\mathbb{P}(\mathcal{A}(x) \in Z) \leq exp(\epsilon \times d(x, x')) \times \mathbb{P}(\mathcal{A}(x') \in Z)$. DP has also been used to mitigate storage side channels [186] and network side channels [187]. In this chapter, I apply this technique to defeat the HPC side channels. This is more challenging as there are more possible leakage sources (e.g., a large number of vulnerable HPC events) and it is hard to control the micro-architectural noise.

4.3 HPC Side Channels

4.3.1 Threat Model

I consider an IaaS cloud scenario protected with encrypted virtualization feature, e.g., AMD SEV, which hence establishes mutual distrust between the customer and the cloud provider. The customer first asks to launch an encrypted VM in the cloud and then perform remote authentication and attestation to confirm if the hardware details and security settings are correct. After the setting confirmation, the sensitive data is sent to the encrypted VM through an encrypted communication channel. As for the cloud provider, I assume it is honest-but-curious, namely it will abide by the defined service agreement but will also seek to gain more sensitive information that it is not explicitly authorized to have. For example, the hypervisor would provide correct register values to guest VMs, including the HPC values. This assumption is realistic for most commercial cloud platforms and has been considered in prior works [87, 92].

While the SEV protection prevents the malicious hypervisor directly extracting the VM's information (e.g., memory, registers, instructions), some micro-architectural side channels can still be constructed to infer the customer's data. In this chapter, I mainly focus on the HPC side channels, where the adversary only needs to monitor HPC events passively, making this attack more stealthy than other active side-channel attacks. Prior works have demonstrated that the adversary can leverage HPCs to accurately extract the victim program's secrets [165–167], making them a severe threat to the confidentiality of trusted computing systems.

4.3.2 Abstraction of HPC Side-channel Attacks

Following previous attacks on various side channels (e.g., power [188], CPU cache [189]), I also model the HPC side-channel attacks as a machine learning task. In the offline stage, the attacker can deploy a template application executing with different secrets $\mathcal{Y}$. Meanwhile, he profiles the application execution behaviors and collects the HPC event leakage traces $\mathcal{X}$. Each trace $x \in \mathcal{X}$ is a time-series of length T, where every time slice $x[t]$ is a vector of monitored events, $1 \leq t \leq T$. Then the attacker trains a parameterized machine learning model f_θ to establish a mapping between $\mathcal{X}$ and $\mathcal{Y}$, i.e., $f_\theta : \mathcal{X} \mapsto \mathcal{Y}$. In the online stage, when the actual victim runs with a secret, the attacker monitors its HPC leakage x, and is able to predict the secret as $y = f_\theta(x)$.

Below I present three practical HPC side-channel attacks against the encrypted VM. The attacks are implemented on an AMD EPYC 7252 processor that supports the SEV protection. The victim VM has the configurations of 4 CPUs, 8G memory and 80G disk, and is launched by the qemu script from AMD [190]. Both the host platform and VM use the Ubuntu 20.04 OS with kernel version 5.11.0. I assume the attacker monitors four HPC events for each attack, as modern processors are usually equipped with four HPC registers. In this demonstration, the events I used are: `RETIRED_UOPS`, `LS_DISPATCH`, `MAB_ALLOCATION_BY_PIPE` and `DATA_CACHE_REFILLS_FROM_SYSTEM`, which cover instruction retirements, operation dispatch and cache accesses. The selection of these HPC events is determined by the ranking results generated by the application profiler, as shown in Section 4.8.1. These four events would leak most information about the secrets sealed in the confidential VM.

4.3.3 Website Fingerprinting Attack

First, I demonstrate an effective website fingerprinting attack (WFA), where the attacker can precisely reveal the website accesses in the encrypted VM. Previous works realized this goal in various scenarios [191–193].

Attack implementation. I design a compact CNN model to fingerprint the website. The model consists of four convolution layers and three fully-connected layers, and also employs common optimizations like batch normalization [194] and

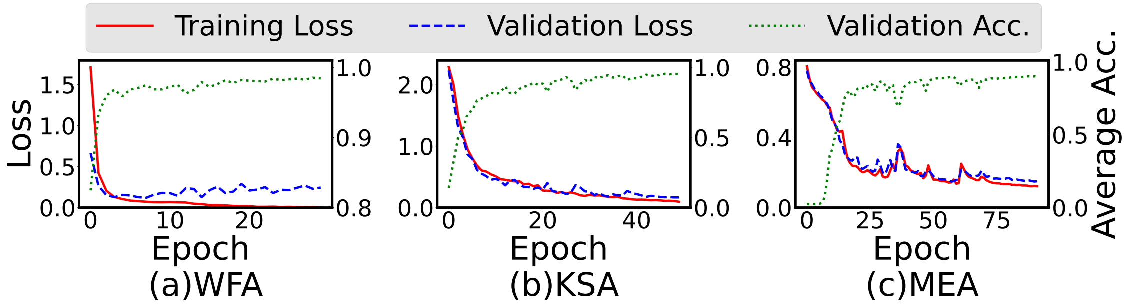


FIGURE 4.1: Training curves of three HPC side-channel attacks.

dropout [195] layers. I select 45 websites from Alexa top-50 websites [196] (excluding 5 blocked websites) as the attack targets, i.e., labels ($\mathcal{Y}$) of the model. The attacker accesses each website using the Chrome browser for 1000 times in the template VM. These websites are accessed in a rolling sequence, which can capture the variants of the sites over time and also avoid IP address blocking by the web servers. At the same time, he uses the host HPCs to sample the counts of four profiled events ($\mathcal{X}$). The sampling process lasts for 3 seconds with an interval of 1ms, giving him a tensor with the size of 4×3000 for each website access. I randomly select 70% and 30% of the dataset for training and validation, respectively.

Attack results. Fig. 4.1a depicts the trends of model accuracy and loss during the training. I can see that the attack accuracy improves very quickly until reaching a stable value (e.g., 98.72%). The well-trained model is then used to predict 4500 actual website accesses (100 accesses for each website) in the victim VM, which achieves an accuracy of 98.57%. It confirms that HPCs can be used to conduct website fingerprinting attacks with high fidelity and efficiency.

4.3.4 Keystroke Sniffing Attack

Another classic side-channel attack is the keystroke sniffing attack (KSA), where the attacker aims to infer the keystrokes entered by the victim user. Past works realized the attack with different side channels (e.g., timing [197], memory [198]).

Attack implementation. As the timing characteristics of keystroke actions can leak information about what those keystrokes are [197], the attack target ($\mathcal{Y}$) is set as the number of keystrokes occurred during the period T, whose timing patterns can be used to infer the keys pressed. The collected leakage trace ($\mathcal{X}$) is the same as WFA. The attacker can also adopt the same CNN model in WFA to predict

keystroke actions. Following the settings in previous works [186, 198], I use the tool `xdotool` [199] to simulate the keystroke actions. It generates K keystrokes in 3 seconds, where K is a random number between $[0, 9]$. This process is repeated 10,000 times in the template VM, where the corresponding HPC event leakage is captured simultaneously. I randomly select 70% for training while the remaining is for validation.

Attack results. Fig. 4.1b shows the training curves of the attacker's model, where the prediction accuracy can finally reach 95.21%. It also shows a very high sniffing accuracy (95.48%) on the test set.

4.3.5 Model Extraction Attack

My last case is the model extraction attack (MEA), which steals the complete neural network architecture of a DNN model. This attack has been realized with the remote-query fashion [200], power [201] and cache [80, 202] side channels.

Attack implementation. The prediction label $\mathcal{Y}$ in MEA is a sequence of layers forming the target DNN model architecture. Hence, this attack should be modeled as a sequence-to-sequence learning task instead of a classification task. I design a RNN model with the CTC decoder [203] to solve this problem. A bidirectional GRU [204] is adopted as the RNN module, as it enables better long-term memory and fully leverages temporal contexts. The best predicted layer sequence is identified with the beam search, which reveals the target model architecture. I select the 30 most commonly used DNN models from the Pytorch official library [205]. I run the inference execution of each model for 1000 times, collect the HPC event sequences, and construct the training and validation sets in a similar way as WFA. Then I train the RNN model to predict the target layer sequence.

Attack results. Fig. 4.1c shows the model training results. The prediction accuracy keeps increasing and finally stays at a stable value (e.g., 91.8%). Note that the accuracy reflects the statistics of the matched layers between prediction and label sequences. The test on the victim VM also shows high attack accuracy (i.e., 90.5%), indicating that the attacker can almost extract the complete architecture of the target DNN model running in the encrypted VM.

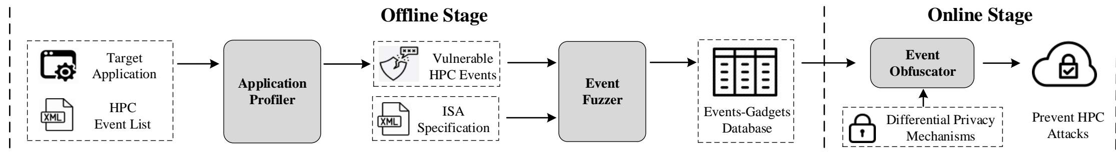


FIGURE 4.2: Overview of my Aegis framework.

4.4 Framework Overview

I aim to design a defense framework for guest users to protect their applications from HPC side channels. A user can only deploy the defense inside his VM, but cannot control the privileged software or hardware on the host server. He does not know which performance counter(s) the attacker will use for information extraction.

I introduce Aegis to achieve my defense goal. The basic idea is to inject noisy instructions into the protected VM's execution flow, which can mask its secret from the HPC events. Aegis has the following benefits. (1) *Unified*: given a protected application, Aegis can mitigate different HPC side channels, regardless of the extraction methodology or performance counters used. (2) *Provable*: I can theoretically guarantee the security of the defense under the given privacy budget. (3) *Automatic*: the entire defense deployment can be achieved automatically without any prior knowledge or human efforts. Fig. 4.2 shows the workflow of Aegis, which consists of three modules in the offline and online stages. The two modules in the offline stage are only performed for one time, and the analyzed results would be applied in the online stage.

Application Profiler (Section 4.5). This module is used to profile the target application with user-specified secrets in the VM, and collect the corresponding leakage of all available HPC events. It adopts information theory to quantify the correlation between the secret data and each event, and identifies the HPC events that are vulnerable as side channels. These events serve as the target for us to defend against.

Event Fuzzer (Section 4.6). This module aims to automatically find out the possible instruction gadgets that can alter the side-channel observations of the HPC events identified from Application Profiler. It takes a machine-readable ISA (Instruction Set Architecture) specification and the list of vulnerable events as the input. With a well-designed grammar model, the fuzzing performs on a significantly

reduced search space and finally outputs the instruction gadgets that can disturb the values of these vulnerable HPC events.

Event Obfuscator (Section 4.7). This module injects the instruction gadgets, selected by Event Fuzzer into the protected VM at runtime, which can effectively mask the HPC values observed by the adversary outside the VM, and prevent side-channel information leakage. To provide a provable security guarantee, I introduce the differential privacy mechanisms to regulate the number of injected instruction gadgets.

4.5 Application Profiler

4.5.1 Challenges

The first step of `Aegis` is to identify possible attack surfaces (e.g., HPC events) that can leak secrets from the target application. There are several challenges to achieve this goal.

C1. Numerous HPC events. A modern processor usually supports a large number of HPC events and any event can be vulnerable to the victim application. For example, in my experiment platforms, the Intel Xeon E5-1650 processor has 6166 usable events while the AMD EPYC 7252 processor has 1903 supported events. While each event should be fuzzed with all possible combinations of instruction sequences (Section 4.6), such numerous events place a heavy burden on the comprehensive analysis of potential leakage sources.

C2. Heterogeneity and Non-determinism. The number and type of available HPC events highly depend on the processor family. Table 4.1 shows the statistics of HPC events from two Intel CPUs and two AMD CPUs. I can see that CPUs from the same family (e.g., Intel E5 family) share similar hardware features (i.e., HPC events), while processors from different families can vary greatly. Besides, it is well known that HPCs cannot provide precise counts of system performance events, because of the external interference, e.g., kernel interactions, hardware interrupts [206]. This introduces extra noise to the profiling process, which may mislead the observation of changes in HPC values.

HPC Statistics	Intel Xeon E5-1650	Intel Xeon E5-4617	AMD EPYC 7252	AMD EPYC 7313P
# of HPC Events	6166	6172	1903	1903
# of Different Events	/	14	/	0

Table 4.1: Statistics of HPC events in various processors.

C3. Vulnerability quantification. Different events lead to various levels of vulnerability to HPC side-channel attacks, i.e., some events can leak more information about the target. Hence, the quantification of event vulnerability is necessary to perform more efficient defense, as I can pay more attentions to those more vulnerable events. However, this issue has never been discussed before and it poses a practical challenge.

4.5.2 Profiling Design

I design an offline Application Profiler module to tackle the above challenges and identify the vulnerable HPC events for a given application executing specified secrets. Basically, I launch a template VM on a template server where I have the host privileges. This server should have a similar processor model (i.e., in the same processor family) as the target cloud server[1] to guarantee the generality of the identified events. Note that the template server can be either a local server or provided by a third-party entity, which has no conflict of interest (e.g., a government agency or a neutral authority). For instance, the guest can rent a bare-metal server from public cloud providers (e.g., AWS or Azure) to serve as the template server, which can have much less resources than the target cloud server and only the processor model needs to be similar. Then I run the target application with a set of customer-specified secrets in the template VM for multiple times and monitor each available HPC event from the host. Finally, the monitored results are automatically analyzed by the program to rank the HPC events based on their vulnerability, i.e., information gain that can be used to extract the application. After the profiling, I can be sure that, at least for the specified secrets, all possible HPC events this application could trigger have been exhaustively activated.

[1]The processor model of the cloud server is obtained from the AMD PSP during the remote attestation.

CPU Processor	Percentage of various event types (%)					
	H	S	HC	T	R	O
Intel Xeon E5-1650	0.39 (100)	0.31 (0)	1.00 (100)	36.15 (7.98)	7.75 (99.37)	54.40 (0)
AMD EPYC 7252	1.26 (100)	1.00 (0)	3.26 (100)	87.17 (1.57)	5.20 (91.83)	2.11 (0)

TABLE 4.2: HPC event distribution, including events of Hardware (H), Software (S), Hardware Cache (HC), Tracepoint (T), Raw CPU (R) and Others (O). Data in brackets shows the percentage remaining after the warm-up profiling.

Monitoring setup. I need to first extract the full list of available HPC events for the given processor model. This can be achieved with a third-party tool `libpfm4` [207]. Then I configure the performance monitoring tool to measure each count of events. In my implementation, I adopt the Linux kernel interface `perf_event_open`, which can effectively reduce the measurement noise as it interacts with the kernel directly. I also set the `pid` and `exclude_kernel` attributes to achieve VM-specific monitoring and prevent influence from the host kernel or other processes. For each time of profiling, I simultaneously monitor four HPC events, which is determined by the upper limit of available HPC registers on the processor. It achieves a suitable trade-off between the monitoring performance and accuracy, as the `perf` subsystem uses time multiplexing [208] for monitoring when there are more monitored events than available registers, which would affect the value accuracy.

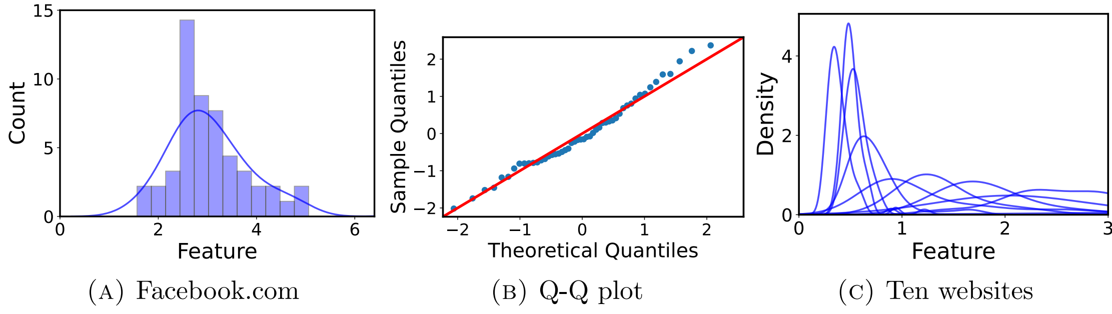
Warm-up profiling. I perform a warm-up profiling to compact the event list and reduce the complexity of vulnerability analysis. The key idea is that a majority of HPC events cannot reflect the activities inside a guest VM. To exclude those events, I measure and compare the event counts when the VM runs the application and when it is idle. The events without any value changes in the counts will be removed from the list, as they cannot reflect the application behaviors. After this warm-up profiling, I only get less than 10% of the events. Take the website fingerprinting analysis as an example, in my two experiment platforms, only 738 (Intel) and 137 (AMD) events remain for further analysis after 5 repeated warm-up profilings, where the results of each profiling are almost the same.

To give more insights on the profiled HPC events, I perform a comprehensive analysis and summarize different types of HPC events in two CPU processors, as shown in Table 4.2. Note that Table 4.2 actually contains all available events

that can be monitored through `perf` subsystem, which include events that are not collected through hardware features, e.g., software (`S`)/tracepoint (`T`) events. But for the sake of description simplicity, they are covered together and also named as HPC events. I observe that the tracepoint events (`T`) and other events (`O`) account for nearly 90% of the total number. The *tracepoint events* measure the access states on the tracepoints provided by the host kernel infrastructure, such as most system calls, most of which cannot precisely capture the application behaviors isolated inside the VM. The *other events* mainly denote the cases at a low level, like hardware breakpoints provided by the CPU, which are generally not be invoked by normal VM applications. I also give the remaining percentage of each event type after the warm-up profiling in Table 4.2, where the remaining events mainly consist of hardware events (`H`/`HC`) and CPU raw events (`R`), while software (`S`)/other (`O`) events and most tracepoint events (`T`) are removed. It indicates that HPC leakages from sealed VM applications are mainly reflected at the hardware level.

Event ranking. My last stage is to profile and rank HPC events based on their vulnerabilities. Given a specific event, assume $\mathcal{Y}$ denote the set of customer-specified secrets executed in the victim application, which contains N_y objects. $\mathcal{X}$ denotes the profiled value traces for the given HPC event, containing $N_x = N_y \times m$ objects, which repeatedly performs m measurements for each secret object. It can average out the non-determined event values.

While the leakage trace $x \in \mathcal{X}$ is a time-series, I first extract the feature value of the sequence with Principle Component Analysis (PCA) [209], which is a widely used feature extraction method for processing high-dimensional data while preserving most of the information. For the sake of simple computation, I follow previous work [197] to fit the monitored event values as a Gaussian-like unimodal distribution. My observation also shows that most event values indeed distributed normally. Fig. 4.3a gives an example distribution over the event `DATA_CACHE_REFILLS_FROM_SYSTEM` on the website *facebook.com*. In Fig. 4.3b, I quantitatively compare the real distribution of the event values to Gaussian distribution $\mathcal{N}(0, 1)$ with the Q-Q plot [210]. The result confirms that the HPC event values of a secret (e.g., a website access) indeed follow the Gaussian distribution. Hence, I can naturally assume that the probability of the event value x between the target secret $y \in \mathcal{Y}$, $P(x|y)$, forms a univariate Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, i.e., $P(x|y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$. Fig. 4.3c shows the estimated distributions of the



(A) Facebook.com (B) Q-Q plot (C) Ten websites

FIGURE 4.3: The distribution of HPC event values.

event values on 10 websites. Although the distributions of some websites overlap slightly, they can still be classified easily, which explains the high attack accuracy of WFA in Section 4.3.

With Gaussian modelling of HPC event values, I can then quantitatively estimate the information gain, i.e., *mutual information*, induced by the given HPC event. First the entropy of the probability distribution of secret y is $H(\mathcal{Y}) = -\sum_{y\in\mathcal{Y}} P(y)logP(y)$. Then, given a event feature value x_0, the entropy of the probability distribution of secret y is $H(\mathcal{Y}|\mathcal{X} = x_0) = -\sum_{y\in\mathcal{Y}} P(y|x_0)logP(y|x_0)$, where $P(y|x_0) = \frac{P(x_0|y)P(y)}{\sum_{y\in\mathcal{Y}} P(x_0|y)P(y)}$. Hence, the *mutual information* can be computed as:

$$I(\mathcal{Y};\mathcal{X}) = H(\mathcal{Y}) - \int P(x_0)H(\mathcal{Y}|\mathcal{X} = x_0)dx_0 \tag{4.1}$$

where $P(x_0) = \sum_{y\in\mathcal{Y}} P(x_0|y)P(y)$. The computed $I(\mathcal{Y};\mathcal{X})$ is the metric to quantify vulnerability of the HPC events.

4.6 Event Fuzzer

For each identified vulnerable HPC event from Section 4.5, `Aegis` calls the offline module Event Fuzzer to search for the instruction sequence gadgets, which can alter the HPC event value, and obfuscate the adversary's side-channel observation. To make this process automatic and efficient, I propose to use the fuzzing technique to find the qualified instructions.

4.6.1 Challenges

There exist a couple of challenges to achieve my goal.

C4. Lack of prior knowledge. Since previous works have never systematically discussed the relationship between instruction execution and changes in HPC counts, I are totally blind to the validity of instructions on obfuscating HPC values. This poses a significant challenge, as I have to fuzz all possible combinations of instructions without knowing their effects ahead, which leads to an infinite search space.

C5. Undesired side effects. The instruction sequence may exhibit unexpected side effects, which could fool the fuzzer and mislead the testing path. For instance, the `store` instruction not only loads the target data into the memory, but also changes the cache state. It creates multiple branches of the code testing, which hence exponentially increases the complexity of the fuzzing process.

C6. Inherited dirty state. To accelerate the search process with high efficiency, all generated instruction gadgets are fuzzed in an uninterrupted way. Unfortunately, such scheme can leave the dirty state of the current gadget to the subsequent ones. For example, following gadgets would inherit the cache state of previous gadgets. Such dirty state entangles successive gadgets, which greatly interferes the fuzzing process and often results in false positive results.

4.6.2 Design Overview

The search of instruction gadgets is modeled as a fuzzing problem. I consider the value change of identified HPC events as a *runtime bug*, and the target is to generate more efficient *inputs* (i.e., instruction gadgets) that can lead to such *bugs*. Given the lack of prior knowledge, using mutation-based fuzzing is not a reasonable choice, as I have no idea about well-performing seeds. To combat that, I adopt grammar-based fuzzing, which however requires a well-designed format model to reduce the search space of inputs.

While the instruction gadget aims to change the HPC event to a specific state, I divide it into two actions, as shown in Fig. 4.4. (1) I first bring the event to a known *reset state* (S_0) with a *reset instruction sequence*. For instance, to monitor

the event of cache references, I issue a `clflush` instruction to empty the cache line. (2) Then a *trigger instruction sequence* is used to transition the event from (S_0) to the desired *trigger state* (S_1), in which the value of monitored HPC event is changed due to the effect of trigger instructions. The combination of the reset and trigger sequences forms the instruction sequence gadget for obfuscating the HPC events, which hence can be considered as the format model for the fuzzing input generation.

Fig. 4.5 shows the workflow of my Event Fuzzer, which consists of the following steps. ① I first clean up the machine-readable ISA specification and remove all illegal instructions for the platform microarchitecture. ② I search for qualified instruction gadgets for the profiled HPC events.The module automatically generates the reset and trigger instruction sequences from the cleaned instruction list, and constructs the gadgets following the fuzzing grammar, i.e., the input format model discussed above. Then it executes the generated gadgets and monitors the value change of the target HPC events. All gadgets that can alter the event value are recorded as the obfuscating factors. ③ I further validate the effectiveness of the identified gadgets with multiple mechanisms, aiming to remove the fuzzing paths invoked by undesired side effects of instructions and mitigate the corner cases caused by the inherited dirty state. ④ Finally I cluster the similar gadgets and filter the best ones for the profiled HPC events. I elaborate the details of each step as below.

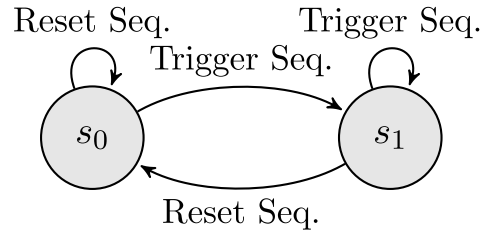


FIGURE 4.4: State transition.

4.6.3 Instruction Cleanup

I first sort out all possible instructions for the target ISA. Note that this is a one-time step, and the cleaned list can be used to fuzze all events. This chapter mainly focuses on the x86 architectures, but the methodology is applicable to other ISA (e.g., ARM) as well. To this end, I obtain a machine-readable x86 instruction list (namely the ISA specification) from uops.info [211]. The list extends each instruction with additional attributes (e.g., effective operand), resulting a large

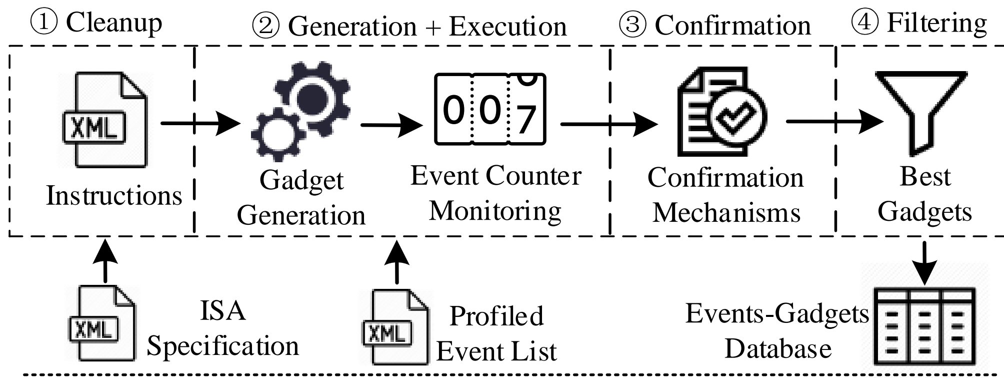


FIGURE 4.5: Workflow of Event Fuzzer.

number of instruction variants. It also gives comprehensive information about each instruction variant, e.g., extension or category, which will be used for filtering in Section 4.6.6.

This ISA specification list contains many illegal instructions which cannot be executed on the given microarchitecture. To remove those instructions, I transfer the ISA specification to an assembly file, and test each instruction. The instructions that cannot complete normally will be excluded from the list. This process significantly reduces the number of instructions in the assembly file. For both Intel and AMD processors, only a small portion (24.16% and 24.31%) of instruction variants are legal. The distribution of faults in the test is similar between two processors, where the majority (98.84% and 98.69%) of the faults are caused by illegal instructions.

4.6.4 Code Generation and Execution

This step aims to generate the instruction gadget that can change the given HPC event to the state S_1. Recall that the instruction gadget consists of a reset sequence and trigger sequence. I randomly sample instructions from the cleaned list to form the sequences, and test their impact on the HPC event. To reduce the fuzzing complexity, I select one instruction for each sequence, and the fuzzing results confirm that this is enough. My methodology can be easily extended to multi-instruction sequences with larger search spaces, which will be considered as future work.

I adopt the `RDPMC` instruction to read HPC values before and after the gadget to measure the corresponding changes. An increased count value indicates that the

gadget may affect the monitoring HPC event. I take several techniques to make the measurement accurate and stable. (1) To reduce the system noise caused by external factors (e.g., interrupts), I properly configure the operating system environment for code execution. By pinning the process to a specific CPU core, I can prevent core transitions from affecting the measurement of HPC values. Besides, I also isolate this entire physical core (e.g., using the Linux kernel parameter `isolcpus`) to ensure that the process is not interrupted by the scheduler. (2) To avoid data corruptions caused by running the instruction gadgets, the code is placed in a dedicated page with the address space between a special prolog and epilog. The prolog saves all callee-saved registers, and creates one page of scratch space on the stack in case some instructions may trash stack values. Furthermore, it initializes all registers that will be used as memory operands to the address of a pre-allocated writable data page. This prevents the corruption of process memory and ensures that executed instructions access the same memory page. The epilog restores the registers and stack state, so that the architectural change can be reverted. (3) To ensure the correct measurement of HPC register values, I inject serializing instructions (e.g., `CPUID`) around the code to regulate the execution flow.

4.6.5 Result Confirmation

This step further validates if a gadget reported by the above step is indeed an obfuscating factor to the given HPC event. I analyze the identified gadgets to eliminate other side effects that can also affect the HPC event values, e.g., unreliable reset sequence whose side effects act as the trigger sequence (Challenge C5) or dirty state inherited from previous executions (Challenge C6). To remove those incorrect gadgets, I propose the following mechanisms.

Multiple executions. As the external factors (e.g., hardware interrupts) can disturb the counts of HPC events, I opt to run the same gadget for multiple times and take the median of measured values. The number of repeated executions sets a trade-off between the fuzzing efficiency and confirmation accuracy: more repetitions increase the confidence of the confirmed results, but also cause longer fuzzing time. In my implementation, I set this parameter to 10, which is proved to be an appropriate value for balancing the trade-off.

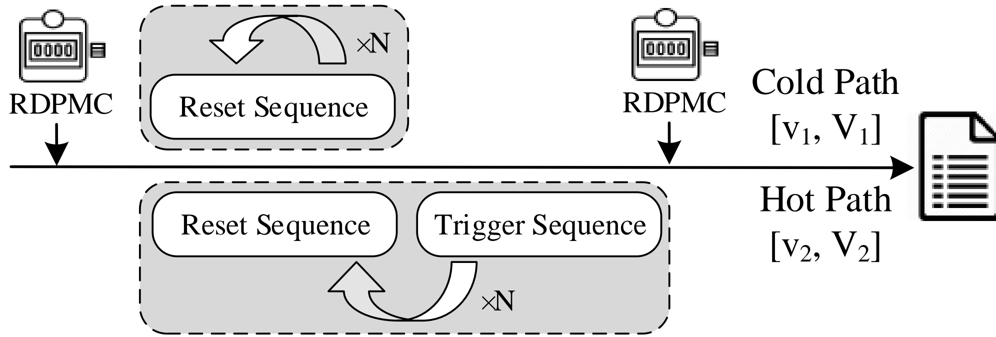


FIGURE 4.6: Execution of repeated triggers.

Repeated triggers. To ensure the HPC value change is indeed caused by the trigger sequence, rather than other undesired side effects of the reset sequence, I also execute the code with only the reset sequence (cold path), in addition to the one with both reset and trigger sequences (hot path), as shown in Fig. 4.6. In each path, I repeat the instruction sequence(s) for R times. The median of the measured count changes is denoted as v_1 and v_2, and the cumulative count changes are denoted as V_1 and V_2, respectively for the cold and hot paths. When these values meet the constraints: $V_2 - V_1 = (1 - \lambda_1)R(v_2 - v_1)$ and $V_2 > \lambda_2 V_1$, I confirm that the event value change is mainly caused by the trigger sequence, i.e., transitioning the state to S_1, and the reset sequence can actually change the state back to S_0 in each execution. My implementation sets λ_1 as a range of [-0.2, 0.2] and $\lambda_2 = 10$.

Gadgets reordering. In order to perform fuzzing as fast as possible, I execute a generated gadget shortly after the previous one, which can cause the dirty state and affect the measurement. To address the issue, I reorder all the gadgets randomly and repeat the executions. I ignore those gadgets that have different behaviors in the reordered test. This can mitigate the influence of repetitive dirty states and remove incorrect candidates with cross-validation.

4.6.6 Gadgets Filtering

For a specific HPC event, there can be different instruction sequence gadgets that can disturb its count value. In practice, certain events like the `Retired Instruction` can be affected by nearly all instruction executions. Hence, I need to filter the confirmed gadgets from the previous step and cluster them into groups with the same

features. This is achieved by analyzing the properties of reset sequences and trigger sequences, including the extension and ISA (e.g., `BASE` or `X87-FPU`) to which the instruction belongs, and the general category (e.g., arithmetic or logical) of the instruction. The intuition of such scheme is that these properties can strongly indicate the root cause of the executed instructions in the underlying microarchitectural level. This step can considerably reduce the number of reported gadgets, and alleviate the burden of the following analysis. Besides, I also extract the gadget that causes the highest value change for each HPC event, as it can lead to larger disturbance to the HPC monitoring with fewer instructions executed.

4.7 Event Obfuscator

4.7.1 Challenges and Insight

To achieve the defense against HPC side channels, I need to tackle two challenges: (1) How to provide provable privacy guarantee for the HPC event obfuscation? (2) How to defeat the attack with minimal introduced noise and performance overhead? The key insight of my methodology is to model the HPC side-channel defense as a differential privacy problem. Previous works have proposed similar methods to mitigate storage side channels [186] and streaming traffic leakages [187], showing the security of systems injected with differential privacy noise can be guaranteed with theoretical proof.

Let x denote the captured HPC leakage trace and $x[t]$ denote a slice of leakage values at time t. Specifically, to prevent information leakage, I change the HPC measurement from $x[t]$ to $\widetilde{x}[t] = x[t] + r_t$, where r_t denotes the random noise following specific distributions. With the injected noise (i.e., instruction sequence gadget) by the victim VM, the malicious hypervisor cannot distinguish the actual behaviors from $\widetilde{x}[t]$. The scale of the random noise r_t can dominate the defense effectiveness, and is also restricted by the VM performance: larger amount of random noise can improve the privacy at the cost of extra performance overhead. Therefore, I leverage the differential privacy principle to theoretically identify the minimal noise under a given privacy budget, which can reduce the impact on the VM performance.

4.7.2 Differential Privacy Mechanisms

I describe two mechanisms to generate the random noise guided by differential privacy.

Laplace Mechanism. This is the most fundamental mechanism for differential privacy, which is achieved by adding controlled noise from the Laplace distribution. As a symmetric version of the exponential distribution, the Laplace distribution can be represented by its Probability Density Function (PDF): $\texttt{Lap}(b) = \frac{1}{2b} exp(-\frac{|x-\mu|}{b})$, where μ is the location and b is the scale. Consider a Laplace distribution with $\mu = 0$, $b = \frac{\triangle_{x[t]}}{\epsilon}$, and $\triangle_{x[t]} = \max_{(x[t],x[t]' \in \mathcal{X})} |x[t] - x[t]'|$, where $x[t]$ and $x[t]'$ are two adjacent series at time slice t. For each sequence x in the monitored HPC sequence set $\mathcal{X}$, the Laplace mechanism computes a noisy sequence $\widetilde{x}$ as follows: $\widetilde{x}[t] = x[t] + r_t, r_t \sim \texttt{Lap}(\frac{\triangle_{x[t]}}{\epsilon})$. For simplicity, I set $\triangle_{x[t]}$ to 1, as the sequence data have been normalized. I have the following theorem:

Theorem 4.1. *The Laplace mechanism $\mathcal{A}(x[t]) = x[t] + \texttt{Lap}(\frac{\triangle_{x[t]}}{\epsilon})$ satisfies ϵ-DP.*

Proof. Let r_t be the noise injected to $x[t]$, i.e., $r_t \sim \texttt{Lap}(\frac{\triangle_{x[t]}}{\epsilon})$. I have

$$\mathbb{P}(\mathcal{A}(x[t]) = Z) = \frac{\epsilon}{2\triangle_{x[t]}} \exp\left(\frac{-\epsilon|r_t - x[t]|}{\triangle_{x[t]}}\right). \tag{4.2}$$

Similarly, $\mathbb{P}(\mathcal{A}(x[t]') = Z) = \frac{\epsilon}{2\triangle_{x[t]}} \exp\left(\frac{-\epsilon|r_t - x[t]'|}{\triangle_{x[t]}}\right)$. Thus,

$$\begin{aligned} \frac{\mathbb{P}(\mathcal{A}(x[t]) = Z)}{\mathbb{P}(\mathcal{A}(x[t]') = Z)} &= \exp\left(\frac{\epsilon(|r_t - x[t]'| - |r_t - x[t]|)}{\triangle_{x[t]}}\right) \\ &\leq \exp\left(\frac{\epsilon(|x[t] - x[t]'|)}{\triangle_{x[t]}}\right) = \exp(\epsilon). \end{aligned} \tag{4.3}$$

□

***d** Mechanism.** This mechanism is extended from Chan et al. [212] and a particular distance metric d^* is used to achieve d-privacy. Assume x and x' are two HPC event sequences, the d^* metric is defined as: $d^*(x, x') = \sum_{t \geq 1} |(x[t] - x[t-1]) - (x'[t] - x'[t-1])|$. Let $\mathbb{N}$ denote the natural numbers and $D(t) \in \mathbb{N}$ denote the largest power of two that divides t, i.e., $D(t) = 2^j$ if and only if $2^j \mid t$ and $2^{j+1} \nmid t$. The d^* mechanism computes the noisy $\widetilde{x}[t]$ as follows: $\widetilde{x}[t] = \widetilde{x}[G(t)] + (x[t] - x[G(t)]) + r_t$,

where

$$G(t) = \begin{cases} 0 & \text{if } t = 1 \\ t/2 & \text{if } t = D(t) \geq 2 \\ t - D(t) & \text{if } t > D(t) \end{cases} \tag{4.4}$$

$$r_t \sim \begin{cases} \texttt{Lap}(\frac{1}{\epsilon}) & \text{if } t = D(t) \\ \texttt{Lap}(\frac{\lfloor log_2 t \rfloor}{\epsilon}) & \text{otherwise} \end{cases} \tag{4.5}$$

Theorem 4.2. *The d^* mechanism satisfies $(d^*, 2\epsilon)$-privacy.*

Proof. It was proven by Xiao et al. [186]. I omit the proof details here. □

Comparisons. The Laplace mechanism is relatively simple and exhibits acceptable privacy guarantee. Furthermore, it suits for a much stricter threat model, where the malicious host even controls and manipulates the calls reading the HPCs. In comparison, for the d^* mechanism, the noise added to each $x[t]$ is closely related to its previous sequences. This intuitively increases the randomness between adjacent sequences, thereby providing better privacy guarantee. However, it is not well suitable for systems requiring high real-time processing speed. Besides, given the limited number of available HPC registers on the processor, the number of concurrently protected events is also restricted. Therefore, d^* mechanism is better suited for reinforcing protection for multiple critical HPC events. I implement both mechanisms in `Aegis` and experimentally discuss their merits and demerits in Sec.5.6. In practical scenarios, customers can choose an appropriate strategy according to the actual system conditions and demands.

4.7.3 Design Details

I implement the online Event Obfuscator as a portable software suite in the victim VM. It is triggered by the cloud customer whenever the critical applications to be protected are launched. Fig. 4.7 shows the workflow of my implementation. It consists of two components: (1) a *kernel module* is used to launch the protection service and monitor the HPC values for the computation of d^*-noise in the d^* mechanism; (2) a *userspace daemon* is used to calculate the amount of random noise and inject it into the execution flow of the VM.

Specifically, the kernel module mainly plays the role of a controller. After receiving the launching signal from the user, it wakes up the userspace daemon. If `Aegis`

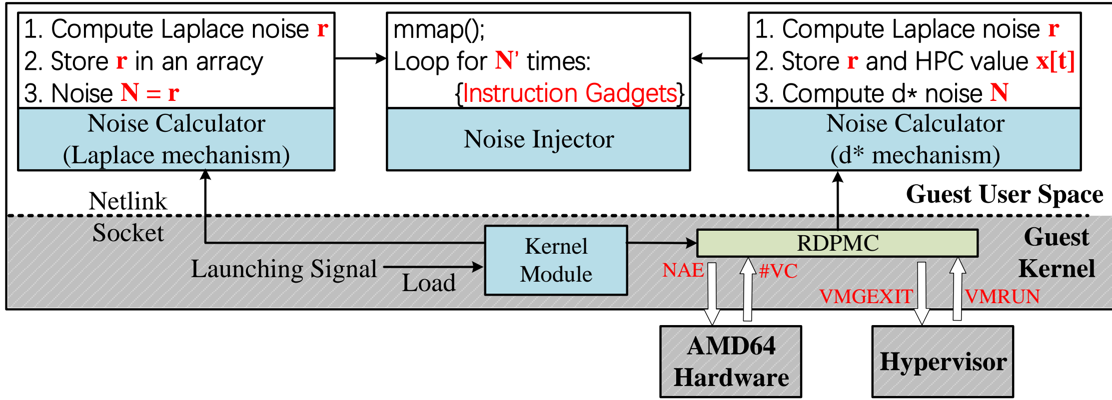


Figure 4.7: Workflow of Event Obfuscator.

adopts the d^* mechanism, the module also needs to monitor the real-time HPC event values x with the `RDPMC` instruction. The recorded HPC values are sent to the userspace daemon with the `netlink` sockets for the subsequent noise generation, which is computation-inefficient in the kernel mode. The userspace daemon consists of two components: noise calculator and noise injector. To support high injection rates, I need to accelerate the calculation of noise value. Hence, the noise calculator maintains a buffer to store the precomputed random noise sequence r following $\texttt{Lap}(\frac{1}{\epsilon})$ for Laplace mechanism or Eq. 4.5 for d^* mechanism. Note that the random number r is directly transferred from the uniform distribution in $[0, 1]$, while using library APIs introduces much longer latency. For the Laplace mechanism, the calculated r is the noise. For the d^* mechanism, HPC values $x[t]$ sent from the kernel module are stored to calculate noise.

After the calculation, the noise injector is called to add the identified amount of noise, i.e., instruction gadgets, into the VM's execution flow. In theory, I can obfuscate each vulnerable HPC events by injecting its corresponding noise gadgets. However, given it usually has hundreds of vulnerable events, such method also introduces hundreds of injected gadgets, which may lead to large performance overhead. In the practical implementation, the identified gadget sets for various HPC events usually have intersections, which allow a gadget to interfere more than one event. For example, gadgets corresponding to `HW_CACHE_L1D:WRITE` can induce changes in almost cache-related events. Therefore, the optimal solution is to extract the smallest gadget set that can cover the most vulnerable events. In my settings, to cover all 137 vulnerable HPC events, I only require 43 instruction gadgets, which hence significantly reduces the overhead invoked by instruction injection. By stacking these gadgets together, I conduct a code segment that executes repeatedly

to inject noise to vulnerable HPC events. The number of repetitions of the code execution is determined by the noise value computed from the noise calculator. This will not affect the original VM execution, and the incurred performance cost is acceptable under a satisfactory privacy budget.

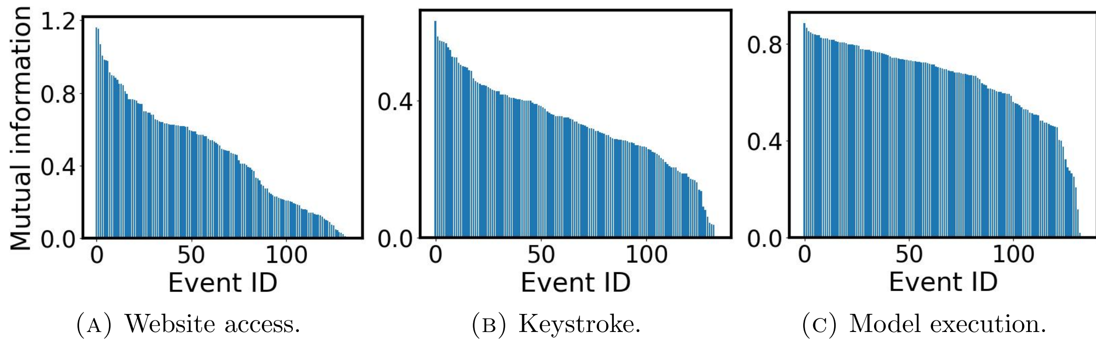
In my implementation, I explicitly pin Event Obfuscator and protected applications to the same virtual CPU core, so that the malicious hypervisor cannot arrange them to different physical cores, and cannot distinguish to bypass my defense. Note that with the protection of SEV, processes on the same virtual CPU core are indistinguishable for the hypervisor even it owns the highest privilege.

4.8 Evaluation

4.8.1 Profiling Evaluation

With the warm-up profiling, I can compact the number of vulnerable HPC events from M to N, where M denotes the number of all available HPC events and N is the number of filtered vulnerable events. In my settings, the value of M is 6166 for Intel CPU and 1903 for AMD CPU. Hence, the time spent on this step is $T_W = (M \times t_w \times 2)/C$, where C denotes the number of available HPC registers that support concurrent monitoring (e.g., $C = 4$ in my testbed), and t_w denotes the monitoring time for each HPC event (e.g., 1 second in my implementations). Note that the profiling of M events should be performed twice to compare their counts. As a result, the warm-up profiling takes 0.85 hours on Intel CPU and 0.26 hours on AMD CPU.

To further estimate the vulnerability of filtered HPC events, I compute the *information gain* induced by the event values to infer the application secret. For each specified secret (i.e., 45 websites, standard keystrokes or 30 DNN models used in Section 4.3) of the target application, given a specific HPC event, the application is repeatedly executed for 100 times to launch the secret, e.g., visiting a selected website. Then I repeat the operation for each HPC event profiled from the warm-up profiling, which finally generates $N \times S \times 100$ leakage traces, where S is the size of the specified secret set. Therefore, the time spent on this profiling step can be calculated as $T_P = (N \times S \times 100 \times t_p)/C$, where t_p is the profiling time for each



(A) Website access. (B) Keystroke. (C) Model execution.

FIGURE 4.8: The mutual information of each HPC event.

event and still 1 second in my settings. Specially, the three target applications in my experiments take 42.81 hours, 9.51hours and 28.54 hours, respectively. Note that this time cost has large space for optimization, e.g., the number of repeated executions can be reduced to 10 times, which is enough for a rough analysis.

The *mutual information* is computed following Eq. 4.1. Fig. 4.8 shows the *mutual information* of each HPC event for website accesses, keystrokes and DNN model executions occurred in an SEV VM. A higher mutual information reflects higher relevance between the event and the victim application, i.e., the event is a more vulnerable attack surface. I can see that Fig. 4.8a and 4.8b drop much faster than Fig. 4.8c, meaning that for the DNN model execution attack there are more vulnerable HPC events. It is because DNN models invoke more interactions with the underlying hardware, e.g., memory accesses and logical calculation, which lead to more HPC leakages.

4.8.2 Fuzzing Evaluation

I run Event Fuzzer on two processors (i.e., Intel Xeon E5-1650 and AMD EPYC 7252) and evaluate the performance of the fuzzing process. For the Intel CPU, 3386 instructions remain after the cleanup step, leading to a total of $3386^2 = 11,464,996$ possible instruction gadgets. These gadgets are repeatedly fuzzed for each profiled HPC event obtained from Application Profiler, thus performing 738 repetitions. A full fuzzing run terminated in 9.3 hours, which results in a throughput of 253,314 gadgets per second. For the AMD processor, while it similarly has $3407^2 = 11,607,649$ usable gadgets, the fuzzing process can be completed in just 2.2 hours, as the processor has much less (i.e., 137) usable HPC events for executing

CPU Processor	Time Consumption (seconds)			
	Cleanup	Generation + Execution	Confirmation	Filtering
Intel Xeon E5-1650	< 1	33210	132	60
AMD EPYC 7252	< 1	7791	29	18

Table 4.3: Time consumption for each fuzzing step.

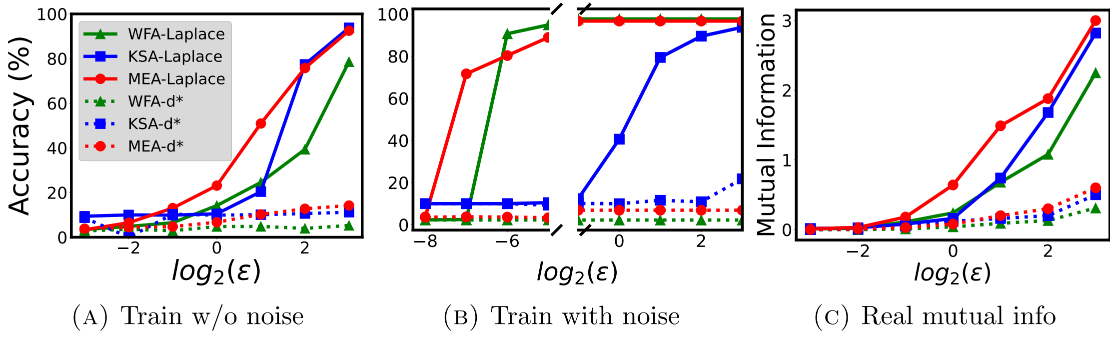
repetitions. The throughput of fuzzing is similar, which reaches 235,449 gadgets per second. Table 4.3 shows the detailed time consumption for each step in the fuzzing process. It can be seen that the generation and execution of gadgets take the most amount of fuzzing time, while other three steps can be achieved in a short time.

After the filtering step, most HPC events only correspond to hundreds or even dozens of usable gadgets, but there are still multiple events corresponding to numerous (e.g., thousands of) gadgets. Availability of more gadgets means I have more choices to obfuscate the HPC event, but it also introduces higher analysis complexity. In the Intel CPU, the mean and median value of the gadgets for all events are 892 and 505, respectively. The event with the most fuzzed gadgets (i.e., 9934) is `MEM_LOAD_UOPS_RETIRED:L1_HIT`. In the AMD CPU, the mean and median are 617 and 440, where the event with the most usable gadgets (i.e., 6219) is `RETIRED_MMX_FP_ INSTRUCTIONS:SSE_INSTR`. It can be seen that events related to the instruction numbers usually tend to be more vulnerable, as they can be modified by most executing instructions. This observation is matched with the profiled results from the above Application Profiler.

Efficiency Analysis. Assuming there are N instructions in the ISA of the target processor, the search space without our event fuzzer is N^{∞}, given that the code gadget could potentially consist of an infinite number of instructions. However, with our fuzzing method, the search space is reduced to M^3, where M represents the number of instructions filtered from the original ISA. Therefore, utilizing fuzzing significantly decreases the search space.

4.8.3 Defense Effectiveness

To evaluate the effectiveness of `Aegis` against HPC side-channel attacks, I vary the privacy budget ϵ, and measure its impact on the attack accuracy. The experiment



(A) Train w/o noise (B) Train with noise (C) Real mutual info

FIGURE 4.9: Impact of ϵ on various attacks.

settings are the same as shown in Section 4.3. As the number of injected instruction gadgets cannot be negative, each noise element is truncated by a clip bound of $[0, B_u]$, where the upper bound B_u is determined empirically based on the profiling of HPC events. For example, I set $B_u = 2e4$ for the event `RETIRED_UOPS`. According to the attacker's capabilities, I consider the following two scenarios:

First, the attacker trains the attack model on clean data collected from the template VM, as the victim VM is a black box for him. Most realistic side-channel attacks follow this case, including my attack cases in Sec.4.3. Figure 4.9a shows the impact of privacy budget ϵ on the accuracy of three attacks for the Laplace and d^* mechanisms. The value of ϵ is set as $[2^{-3}, 2^{-2}, ..., 2^3]$. From this figure, I can summarize four remarks: (1) both mechanisms can effectively mitigate HPC side-channel attacks, which decrease the attack accuracy from $> 90\%$ to 2%; (2) a larger ϵ leads to a higher attack accuracy, since it adds less noise; (3) with the same ϵ, d^* mechanism can provide stronger privacy guarantee, especially for large ϵ (e.g., $\epsilon \geq 2^0$); (4) WFA and KSA are more sensitive to the noise than MEA, as their accuracy decrease much faster with the increase of noise. It may be because website accesses and keystrokes have more similar leakage patterns that are easier to be affected by the injected noise.

Second, I consider a more powerful attacker, who knows the defense details adopted in the VM (e.g., the privacy mechanism, the value of ϵ). In this case, the attacker can train his attack model with noisy data to increase the model's robustness in the exploitation phase. Figure 4.9b shows the attack accuracy of such models under the defense with my **Aegis**, where $\epsilon \in [2^{-8}, 2^{-7}, ..., 2^3]$. I observe that d^* mechanism can still defeat these advanced attacks well, while Laplace mechanism requires a smaller ϵ to suppress the attack accuracy. I conclude that by slightly decreasing

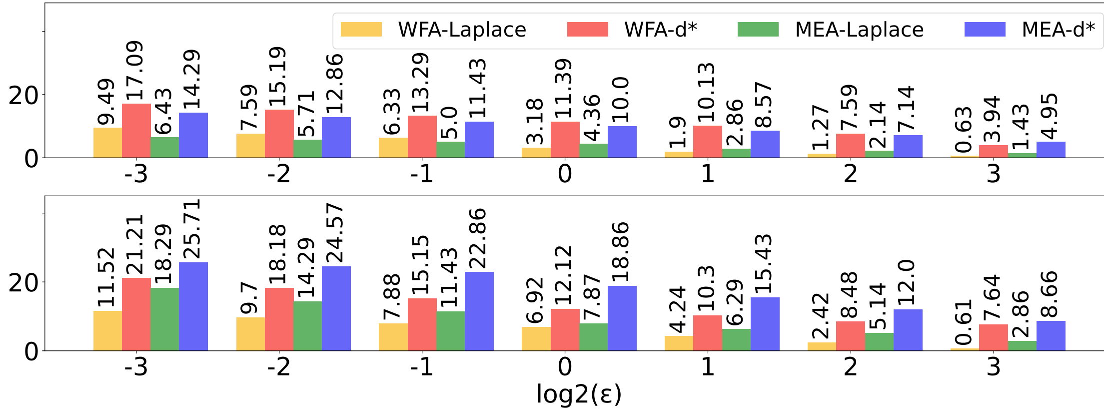


FIGURE 4.10: Impact of ϵ on the latency overhead (upper) and CPU usage (lower).

the privacy budget, `Aegis` can still mitigate HPC attacks even when the attack model is trained in a more robust approach. Given it needs to inject more noise, the defense would induce much larger overhead. However, as such an attacker is too strong and is nearly impossible in the practical scenario, I would not consider it in my following efficiency analysis.

Note that my defense is effective for all machine learning based attack models, as the correlation between the HPC side channels and the running secrets are significantly reduced. Fig. 4.9c shows the value of real mutual information $I(\mathcal{X};\mathcal{X}')$ between the clean HPC leakage traces $\mathcal{X}$ and the noised HPC leakage traces $\mathcal{X}'$ under different size of noise. It can be seen that with the increased noise (i.e., smaller ϵ), the value of $I(\mathcal{X};\mathcal{X}')$ keeps decreasing to a small value. Hence, the mutual information $I(\mathcal{X}';\mathcal{Y})$ between the noised trace $\mathcal{X}'$ and the secret $\mathcal{Y}$ also decreases equivalently [213]. Therefore, although I only show three attack cases in this chapter, the effectiveness of my method can be well guaranteed on other attacks.

4.8.4 Defense Efficiency

As keystrokes are transient actions that only take negligible resources, I mainly focus on the defense against WFA and MEA. I evaluate the efficiency of `Aegis` from latency overhead and CPU usage. Figure 4.10 shows the overhead invoked by various mechanisms.

First, I assess the impact of `Aegis` on the performance of the protected applications in the VM. For each test, I continually access 45 websites or run 30 DNN models to measure the execution time. Figure 4.10 shows the average time of loading a website and running a model inference under two DP mechanisms with different ϵ values in `Aegis`. The website loading time is recorded by the built-in development tool in the Chrome browser, while the DNN model inference time is measured by a timer written in python. From the figure, I can find that (1) smaller ϵ leads to longer execution time, as more noisy instructions are injected; (2) under the same ϵ, d^* mechanism induces more overhead than Laplace mechanism. To guarantee privacy against attacks, I select $\epsilon = 2^0$ for Laplace mechanism (marked with shadow), which causes 3.18% and 4.36% overhead on the execution time of website accesses and model inferences. For d^* mechanism, $\epsilon = 2^3$ is enough to mitigate attacks, where the execution time increases by 3.94% and 4.95% for the two applications. Hence, the time overhead introduced by `Aegis` is slight.

Second, I measure the impact of `Aegis` on the VM resource consumption by monitoring its CPU utilization. As `Aegis` injects instruction gadgets into the VM's executions, it may consume extra CPU resources with certain overhead. According to the basic feature of differential privacy, i.e., the statistical characteristics of noisy result $\widetilde{x}$ are similar to the original data x, I speculate such cost penalty caused by `Aegis` should be small. Figure 4.10 shows the CPU usage of the victim VM under two DP mechanisms. The VM's CPU usage is measured from the host with the `top` tool every 0.2 seconds and each usage value is the average of 5 experiments. Specifically, `Aegis` has smaller influence on the website accesses, which may be because they involve fewer CPU interactions. With Laplace mechanism, the CPU usage penalty introduced by `Aegis` is 6.92% and 7.87% for website accesses and model inferences. For d^* mechanism, it is 7.64% and 8.66%. Hence, `Aegis` only leads to a small CPU overhead.

4.9 Discussion

4.9.1 Alternative Defense Strategies

Constant HPC output. Setting the HPC output to a constant can also mask HPC side channels. However, such method is actually impractical in the real

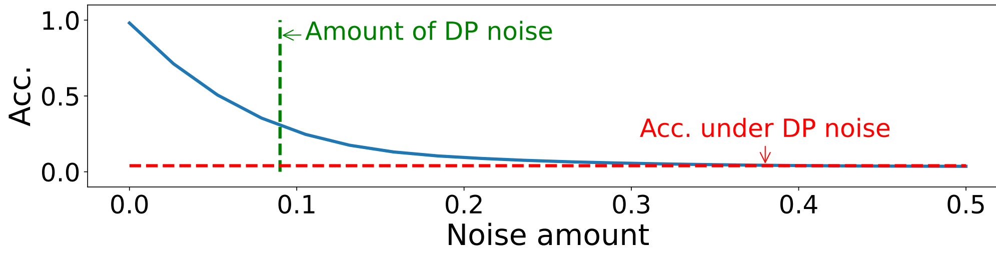


FIGURE 4.11: Attack accuracy with the random noise

implementation. To make the HPC output as a constant, I must add more counts (i.e., noise) to the original HPC values, until reaching the peak HPC value p in the leakage trace. It introduces much more noise than my solution. For example, to obfuscate the leakage of accessing *www.youtube.com*, setting the HPC event (e.g., `DATA_CACHE_REFILLS_FROM_SYSTEM`) values to a constant p totally introduces 595,371,616 event counts, while my Laplace mechanism only introduces 33,090,214 events. Hence, constant HPC output invokes nearly 18× more noise, which is an overkill defense.

Random noise. Instead of the DP noise, simply adding random noise can also obfuscate the HPC leakage. However, this strategy has two limitations. First, such random noise cannot provide a provable privacy guarantee, which still exposes the encrypted VM to the leakage threat. Second, this strategy usually introduces more noise than DP mechanisms to achieve the same privacy protection. I use different scales of random noise to obfuscate the HPC leakage and the attack accuracy is shown in Fig. 4.11. The x-axis denotes the upper bound of random noise in the range of $[0, 0.5] \times p$, where p denotes the peak HPC value in the leakage trace. In the figure, I also label the amount of Laplace noise ($\epsilon = 2^0$) required to effectively defeat the attack, i.e., decreasing the attack accuracy to $< 5\%$. With the same amount of injected noise, random noise mechanism can only decrease the attack accuracy to 32%, which is much higher than DP mechanisms. Besides, I also show the attack accuracy under the effective defense with the DP noise. To achieve the same accuracy, the upper bound of random noise needs to be at least $0.4p$, which introduces 4.37× more noise than the Laplace mechanism.

Isolating guest HPCs. The root cause of HPC side channels is the sharing of HPC registers between the guest and host. Hence, isolating guest HPCs from the malicious host can fundamentally eliminate such side channels, as demonstrated by Intel TDX. However, since this defense necessitates specific hardware modifications,

it is not feasible for existing SEV-based systems, which motivates my software-based solution. While my work can effectively and efficiently mitigate HPC side channels on SEV VMs, it still introduces extra overhead and is not the optimal solution for defending such hardware side channels. Hence, I advocate for AMD to enhance their hardware design, which is a promising alternative of my software design to address this issue from the root.

4.9.2 Analysis with Multiple Tries

The injected noise to the HPC leakage traces actually can be averaged out by the attacker by obtaining a group of leakage traces corresponding to the same secret [213]. However, in the practical scenario, the adversarial hypervisor cannot force the user VM to repeatedly run the same secret for many times. Hence, the adversary cannot collect multiple leakage traces with the same secret to remove the impact of injected noise. Besides, even if the adversary can collect such multiple traces, the attack can be easily defeated by attaching a constant secret-dependent noise to the execution so that the adversary cannot average out the injected noise through analyzing multiple leakage traces. Such method can well protect the user secrets and also reduce the overhead of noise generation.

4.10 Conclusion

In this chapter, I propose `Aegis`, a unified framework that can mitigate confidential VMs from HPC side-channel attacks with a provable privacy guarantee and minimal performance overhead. It also comprehensively profiles the vulnerability of HPC events, and automatically demystifies the correlations between the specific instruction gadgets and HPC event statistics. My future works aim to study the defense effect of noise gadgets with more instructions, and investigate the effectiveness of `Aegis` on more fine-grained attacks, e.g., stealing cryptographic keys. Besides, I also intend to generalize my framework to more micro-architectural attacks, e.g., cache and memory side channels, Meltdown [75] and Spectre [214] attacks, or the latest voltage glitching attack [215].

Part II

New Designs for Confidential Computing with Emerging Applications

Chapter 5

Ownership Verification of DNN Architectures via Hardware Cache Side Channels

Deep Neural Networks (DNN) are gaining higher commercial values in computer vision applications, e.g., image classification, video analytics, etc. This calls for urgent demands of the intellectual property (IP) protection of DNN models. However, as described in Chapter 3, even the confidential TEE sandbox cannot entirely eliminate the information leakage of sealed IP models. In this chapter, I integrate side-channel analysis to propose a benign design suitable for adoption in confidential AI systems, i.e., a novel watermarking scheme to achieve the ownership verification of DNN architectures. Existing watermarks all embedded watermarks into the model parameters while treating the architecture as public property. These solutions were proven to be vulnerable by an adversary to detect or remove the watermarks. In contrast, I claim the model architectures as an important IP for model owners, and propose to implant watermarks into the architectures. I design new algorithms based on Neural Architecture Search (NAS) to generate watermarked architectures, which are unique enough to represent the ownership, while maintaining high model usability. Such watermarks can be extracted via side-channel-based model extraction techniques with high fidelity, allowing it to be deployed with conventional confidential computing systems to provide stronger security guarantee. I conduct comprehensive experiments on watermarked CNN models for image classification tasks and the experimental results show my scheme has negligible impact

on the model performance, and exhibits strong robustness against various model transformations and adaptive attacks.

5.1 Introduction

Deep Neural Networks (DNNs) have shown tremendous progress to solve artificial intelligence tasks. Novel DNN algorithms and models were introduced to interpret and understand the open world with higher automation and accuracy, such as image processing [216–218], video processing [219, 220], natural language processing [221, 222], bioinformatics [223]. With the increased complexity and demand of the tasks, it is more costly to generate a state-of-the-art DNN model: design of the model architecture and algorithm requires human efforts and expertise; training a model with satisfactory performance needs a large amount of computation resources and valuable data samples. Hence, commercialization of the deep learning technology has made DNN models the core Intellectual Property (IP) of AI products and applications.

Release of DNN models can incur illegitimate plagiarism, unauthorized distribution or reproduction. Therefore, it is of great importance to protect the IP of such valuable assets. Similar to image watermarking [224–230], one common approach for IP protection of DNN models is DNN watermarking, which processes the protected model in a *unique* way such that its owner can recognize the ownership of his model. Existing solutions all implanted the watermarks into the parameters for ownership verification [231–236]. The watermark also needs to guarantee satisfactory performance for the protected model. For example, Adi et al. [233] embedded backdoor images with certain trigger patterns into image classification models for IP protection.

Unfortunately, those parameter-based watermarking solutions are not practically robust. An adversary can easily defeat them without any knowledge of the adopted watermarks. First, since these schemes modify the parameters to embed watermarks, the adversary can also modify the parameters of a stolen model to remove the watermarks. Past works have designed such watermark removal attacks, which leverage model fine-tuning [237–239] or input transformation [240] to successfully invalidate existing watermark methods. Second, watermarked models need to give

unique behaviors, which inevitably make them detectable by the adversary. Some works [241, 242] introduced attacks to detect the verification samples and then manipulate the verification results.

Motivated by the above limitations, I propose a fundamentally different watermarking scheme. Instead of protecting the parameters, I treat the network architecture as the IP of the model. There are a couple of incentives for the adversary to plagiarize the architectures [80, 123]. First, it is costly to craft a qualified architecture for a given task. Architecture design and testing require lots of valuable human expertise and experience. Automated Machine Learning (AutoML) is introduced to search for architectures [121], which still needs a large amount of time, computing resources and data samples. Second, the network architecture is critical in determining the model performance. The adversary can steal an architecture and apply it to multiple tasks with different datasets, significantly improving the financial benefit. In short, "the industry considers top-performing architectures as intellectual property" [123], and "obtaining them often has high commercial value" [80]. *Therefore, it is worthwhile to treat the architecture design as an important IP and provide particular protection to it.*

I aim to design a methodology to generate unique network architectures for the owners, which can serve as the evidence of ownership. This scheme is more robust than previous solutions, as maliciously refining the parameters cannot tamper with the watermarks. The adversary can only *remarkably* change the network architecture with large amounts of resources and effort in order to erase the watermarks. This will not violate the copyright, since the new architecture is totally different from the original one, and can be legally regarded as the adversary's own IP. Two questions need to be answered in order to establish this scheme: *(1) how to systematically design architectures, that are unique for watermarking and maintain high usability for the tasks? (2) how to extract the architecture of the suspicious model, and verify the ownership?*

I introduce a set of techniques to address these questions. For the first question, I leverage Neural Architecture Search (NAS) [121]. NAS is a very popular AutoML approach, which can automatically discover a good network architecture for a given task and dataset. A quantity of methods [129, 130, 132, 134, 243, 244] have been proposed to improve the search effectiveness and efficiency, and the searched architectures can significantly outperform the ones hand-crafted by humans. Inspired

by this technology, I design a novel NAS algorithm, which fixes certain connections with specific operations in the search space, determined by the owner-specific watermark. Then I search for the rest connections/operations to produce a high-quality network architecture. This architecture is unique enough to represent the ownership of the model (Section 5.4).

The second question is solved by cache side-channel analysis. Side-channel attacks are a common strategy to recover confidential information from the victim system without direct access permissions. Recent works designed novel attacks to steal DNN models [80, 81, 83]. My scheme applies such analysis for IP protection, rather than confidentiality breach. The model owner can use side-channel techniques to extract the architecture of a black-box model to verify the ownership, even the model is encrypted or isolated inside a confidential TEE sandbox. It is difficult to directly extend prior solutions [79, 80] to my scenario, because they are designed only for conventional DNN models, but fail to recover new operations in NAS. I devise a more comprehensive method to identify the types and hyper-parameters of these new operations from a side-channel pattern. This enables us to precisely extract the watermark from the target model (Section 5.5).

The integration of these techniques leads to the design of my watermarking framework. Experiments on DNN models for image classification show that my method is immune to common model parameter transformations (fine-tuning, pruning), which could compromise prior solutions. Furthermore, I test new adaptive attacks that moderately refine the architectures (e.g., shuffling operation order, adding useless operations), and confirm their incapability of removing the watermarks from the target architecture. In sum, I make the following contributions:

- It is the *first* work to protect the IP of DNN architectures. It creatively uses the NAS technology to embed watermarks into the model architectures.
- It presents the *first* positive use of cache side channels to extract and verify watermarks, further enhancing the security guarantee of confidential AI systems.
- It gives a comprehensive side-channel analysis about sophisticated DNN operations that are not analyzed before.

5.2 Related Works on DNN Watermarking

Numerous watermarking schemes have been proposed for conventional DNN models. They can be classified into the following two categories:

5.2.1 White-box solutions

This strategy adopts redundant bits as watermarks and embeds them into the model parameters. For instance, Uchida et al. [231] introduced a parameter regularizer to embed a bit-vector (e.g. signature) into model parameters which can guarantee the performance of the watermarked model. Rouhan et al. [232] found that implanting watermarks into model parameters directly could affect their static properties (e.g histogram). Thus, they injected watermarks in the probability density function of the activation sets of the DNN layers. These methods require the owner to have white-box accesses to the model parameters during the watermark extraction and verification phase, which can significantly limit the possible usage scenarios.

5.2.2 Black-box solutions

This strategy takes a set of unique sample-label pairs as watermarks and embeds their correlation into DNN models. For examples, Le et al. [245] adopted adversarial examples near the frontiers as watermarks to identify the ownership of DNN models. Zhang et al. [246] and Adi et al. [233] employed backdoor attack techniques to embed backdoor samples with certain trigger patterns into DNN models. Namba et al. [241] and Li et al. [247] generated watermark samples that are almost indistinguishable from normal samples to avoid detection by adversaries.

Different from these works, I propose a new watermarking scheme. Instead of modifying the parameters, my approach makes the architecture design as Intellectual Property, and adopts cache side channels for architecture verification. This strategy can defeat all the watermark removal attacks via parameter transformations.

5.3 Preliminaries

5.3.1 Definition of A NAS Method

In this chapter, I mainly focus on NAS methods using the cell-based search space, as it is the most popular and efficient strategy. Formally, I consider a NAS task, which aims to construct a model architecture containing N cells: $\mathfrak{A} = \{c_1, ..., c_N\}$. The search space of each cell is denoted as $S = (\mathcal{G}, \mathcal{O})$. $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is the DAG representing the cell *supernet*, where set $\mathcal{N}$ contains two inputs (a, b) from previous cells and $\mathcal{B}$ computing nodes in the cell, i.e., $\mathcal{N} = \{a, b, \mathcal{N}_1, ..., \mathcal{N}_\mathcal{B}\}$; $\mathcal{E} = \{\mathcal{E}_1, ..., \mathcal{E}_B\}$ is the set of all possible edges between nodes and $\mathcal{E}_j$ is the set of edges connected to the node $\mathcal{N}_j$ $(1 \leq j \leq B)$. Each node can only sum maximal two inputs from previous nodes. $\mathcal{O}$ is the set of candidate operations on these edges. Then I combine the search spaces of all cells as $\mathbb{S}$, from which I try to look for an optimal architecture $\mathfrak{A}$. The NAS method is defined as below:

Definition 5.1. (NAS) A NAS method is a machine learning algorithm that iteratively searches optimal cell architectures from the search space $\mathbb{S}$ on the proxy dataset $\mathcal{D}$. These cells construct one architecture $\mathfrak{A} = \{c_1, ..., c_N\}$, i.e., $\mathfrak{A} = \textbf{\textit{NAS}}(\mathbb{S}, \mathcal{D})$.

After the search process, $\mathfrak{A}$ is trained from the scratch on the task dataset $\overline{\mathcal{D}}$ to learn the optimal parameters. The architecture $\mathfrak{A}$ and the corresponding parameters give the final DNN model $f = \texttt{train}(\mathfrak{A}, \overline{\mathcal{D}})$.

5.3.2 Definition of A Watermarking Scheme

A watermarking scheme for NAS enables the ownership verification of DNN models searched from a NAS method. This is formally defined as below:

Definition 5.2. A watermarking scheme for NAS is a tuple of probabilistic polynomial time algorithms (***WMGen***, ***Mark Verify***), where

- ***WMGen*** takes the search space of a NAS method as input and outputs secret marking key mk and verification key vk.

- ***Mark*** outputs a watermarked architecture $\mathfrak{A}$, given a NAS method, a proxy dataset $\mathcal{D}$, and mk.
- ***Verify*** takes the input of vk and the monitored side-channel trace, and outputs the verification result of the watermark in $\{0, 1\}$.

A strong watermarking scheme for NAS should have the following properties [233, 246].

Effectiveness. The watermarking scheme needs to guarantee the success of the ownership verification over the watermarked $\mathfrak{A}$ using the verification key. Formally,

$$Pr[\textbf{Verify}(vk, \mathbb{T}) = 1] = 1, \tag{5.1}$$

where $\mathbb{T}$ is the monitored side-channel trace from $\mathfrak{A}$.

Usability. let $\mathfrak{A}_0$ be the original architecture without watermarks. For any data distribution $\mathcal{D}$, the watermarked architecture $\mathfrak{A}$ should exhibit competitive performance compared with $\mathfrak{A}_0$ on the data sampled from $\mathcal{D}$, i.e.,

$$|Pr[f_0(x) = y|(x, y) \sim \mathcal{D}] - Pr[f(x) = y|(x, y) \sim \mathcal{D}]| \leq \epsilon. \tag{5.2}$$

where $f = \texttt{train}(\mathfrak{A}, \overline{\mathcal{D}})$ and $f_0 = \texttt{train}(\mathfrak{A}_0, \overline{\mathcal{D}})$.

Robustness. Since a probabilistic polynomial time adversary may modify f moderately, I expect the watermark remains in $\mathfrak{A}$ after those changes. Formally, let $\mathbb{T}'$ be the side-channel leakage of a model f' transformed from f, where f' and f are from the same architecture $\mathfrak{A}$ with similar performance. I have

$$Pr[\textbf{Verify}(vk, \mathbb{T}') = 1] \geq 1 - \delta \tag{5.3}$$

Uniqueness. A normal user can follow the same NAS method to learn a model from the same proxy dataset. Without the marking key, the probability that this common model contains the same watermark should be smaller than a given threshold δ. Let $\mathbb{T}'$ be the side channel leakage of a common model learned with the same dataset and NAS method, I have

$$Pr[\textbf{Verify}(vk, \mathbb{T}') = 1] \leq \delta. \tag{5.4}$$

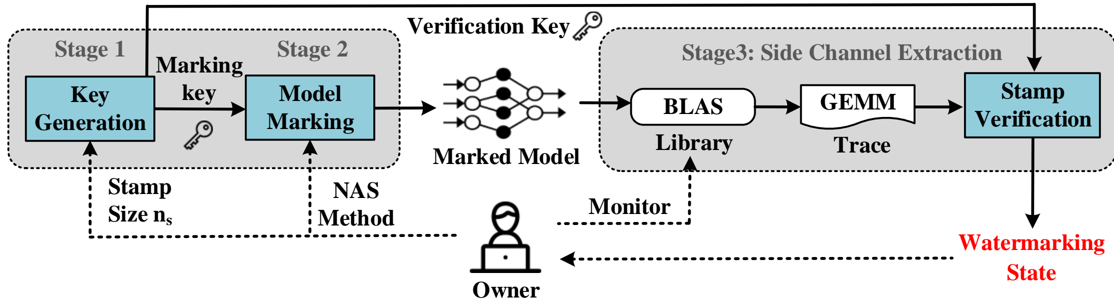


FIGURE 5.1: Overview of my watermarking framework

5.4 My Watermarking Scheme

Figure 5.1 shows the overview of my watermarking framework, which consists of three stages. At stage 1 , the model owner generates a unique watermark and the corresponding key pair (mk, vk) using the algorithm **WMGen** (Section 5.4.1). At stage 2, he adopts a conventional NAS method with the marking key mk to produce the watermarked architecture following the algorithm **Mark** (Section 5.4.2). He then trains the model from this architecture. Stage 3 is to verify the ownership of a suspicious model: the owner collects the side-channel information at inference, and identifies any potential watermark based on the verification key vk using the algorithm **Verify** (Section 5.4.3). Below I describe the details of each stage, followed by a theoretical analysis (Section 5.4.4).

5.4.1 Watermark Generation (WMGen)

According to Definition 5.1, a NAS architecture is a composition of cells. Each NAS cell is actually a sampled sub-graph of the *supernet* $\mathcal{G}$, where the attached operations are identified by the search strategy. To generate a watermark, the model owner selects some edges from $\mathcal{G}$ which can form a path. I select the edges in a path because the executions of their operations have dependency (see the red edges in Figure 5.5). So an adversary cannot remove the watermarks by shuffling the operation order at inference. Then the model owner fixes each of these edges with a randomly chosen operation. The set of the fixed edge-operation pairs $\{s_e : s_o\}$ inside a cell is called a *stamp*, as defined below:

Definition 5.3. (Stamp) A stamp for a cell is a set of edge-operation pairs $\{s_e : s_o\}$, where s_e, s_o denote the selected edges in a path and the corresponding operations, respectively.

The combination of the stamps of all the cells form a watermark for a NAS architecture:

Definition 5.4. (Watermark) Consider a NAS method with a proxy dataset $\mathcal{D}$ and search space $\mathbb{S}$. $\mathfrak{A} = \{c_1, ..., c_N\}$ represents the neural architecture produced from this method. A watermark for $\mathfrak{A}$ is a set of stamps $mk_1, ..., mk_N$, where mk_i is the stamp of cell c_i.

Algorithm 2 Marking Key Generation (**WMGen**)

Input: # of fixed edges n_s, search space $\mathbb{S} = (\mathcal{G}, \mathcal{O})$

Output: marking key mk, verification key vk

$S_e =$ `GetPath`$(\mathcal{G}, n_s)$ **for** i from *1 to N* **do**

  $s_e \leftarrow$ randomly select one path from S_e

  $s_o \leftarrow$ randomly select n_s operations from $\mathcal{O}$ for s_e

  $mk_i = \{s_e : s_o\}$, $vk_i = s_o$

return $mk = (mk_1, ..., mk_N)$, $vk = (vk_1, ..., vk_N)$

Algorithm 2 illustrates the detailed procedure of constructing a watermark and the corresponding marking and verification keys (mk, vk). Given the supernet $\mathcal{G}$, I call function `GetPath` to obtain a set S_e of all the possible paths with length n_s, where n_s is the predefined number of stamp edges ($1 \leq n_s \leq \mathcal{B}$). Then for each cell c_i, I randomly sample a path s_e from S_e. Edges in the selected path are attached with fixed operations s_o chosen by the model owner to form the cell stamp $mk_i = \{s_e : s_o\}$. Finally I can construct a marking key $mk = (mk_1, ..., mk_N)$. The verification key is $vk = (vk_1, ..., vk_N)$, where vk_i is the fixed operation sequence s_o in cell c_i.

In my implementation, I randomly sample the paths and operations for the marking key. It is also possible the model owner crafts the stamps based on his own expertise. He needs to ensure the design is unique and has very small probability to conflict with other models from the same NAS method. I do not discuss this option in this chapter.

5.4.2 Watermark Embedding (Mark)

To generate a competitive DNN architecture embedded with the watermark, I fix the edges and operations in the marking key mk, and apply a conventional NAS method to search for the rest connections and operations for a good architecture. This process will have a smaller search space compared to the original method. However, as shown in previous works [129, 132], there are multiple sub-optimal results with comparable performance in the NAS search space, which makes random search also feasible. Hence, I hypothesize that I can still find out qualified results from the reduced search space. Evaluations in Section 5.6 verify that the reduced search space incurs negligible impact on the model performance.

Algorithm 3 shows the procedure of embedding the watermark to a NAS architecture. For each cell c_i in the architecture, I first identify the fixed stamp edges and operations $\{s_e : s_o\}$ from key mk_i. Then the cell search space S is updated as $(\mathcal{G} = (\mathcal{N}, \overline{\mathcal{E}}), \mathcal{O})$, where $\overline{\mathcal{E}}$ is the set of connection edges excluding those fixed ones: $\overline{\mathcal{E}} = \mathcal{E} - s_e$. The updated search spaces of all the cells are combined to form the search space $\mathbb{S}$, from which the NAS method is used to find a good architecture $\mathfrak{A}$ containing the desired watermark.

Algorithm 3 Watermark Embedding (**Mark**)

Input: marking key mk, `NAS` method, proxy dataset $\mathcal{D}$

Output: watermarked architecture $\mathfrak{A}$

$\mathbb{S} \leftarrow$ search space of the whole model

for each cell c_i **do**

  retrieve $\{s_e : s_o\}$ from mk_i

  $\overline{\mathcal{E}} = \mathcal{E} - s_e$; $S = (\mathcal{G} = (\mathcal{N}, \overline{\mathcal{E}}), \mathcal{O})$

  $\mathbb{S}$.append(S)

$\mathfrak{A} = \texttt{NAS}(\mathbb{S}, \mathcal{D})$

return $\mathfrak{A}$

Discussion. I describe my watermarking scheme with the NAS technique. It is worth noting that my methodology can also be applied to the hand-crafted architectures. The model owner only needs to inject the stamp edges to some locations inside his designed architecture and then train the model. I consider the evaluation of this strategy as future work.

5.4.3 Watermark Verification (Verify)

During verification, I utilize cache side channels to capture an execution trace $\mathbb{T}$ by monitoring the inference process of the target model M'. Details about side-channel extraction can be found in Section 5.5. Due to the existence of extra computations like concatenating and preprocessing, cells in $\mathbb{T}$ are separated by much larger time intervals and can be identified as sequential leakage windows. If $\mathbb{T}$ does not have observable windows, I claim it is not generated by a cell-based NAS method and is out of the consideration. A leakage window further contains multiple clusters, each of which corresponds to an operation inside the cell.

Algorithm 4 describes the verification process. First the side-channel leakage trace $\mathbb{T}$ is divided into cell windows, and for the i-th window, I retrieve its stamp operations s_o from vk_i. Then the cluster patterns in the window are analyzed in sequence. Since the adversary can possibly shuffle the operation order or add useless computations to obfuscate the trace, *I only verify if the stamp operations exist in the cell in the correct order, which is not affected by the obfuscations due to their execution dependency, while ignoring other operations.* Besides, since the adversary may inject useless cell windows to obfuscate the verification, I only consider cells that contain the expected side-channel patterns and skip other cells. Once the number of verified cells is equal to the size of generated verification key, I can claim the architecture ownership of the DNN model.

Algorithm 4 Watermark Verification (**Verify**)

Input: verification key vk, monitored trace $\mathbb{T}$, # of fixed edges n_s
Output: verification result

```
Split 𝕋 into cell windows, go_on = 1, verified_wins = 0
for each window_i in 𝕋 do
    if go_on == 1 then
        retrieve s_o from vk_i, id ← 0
    for each cluster in window_i do
        if match(cluster, s_o[id]) = True then
            id+ = 1
    if id == n_s then
        go_on = 1, verified_wins += 1
    else
        go_on = 0
return (verified_wins == vk.size()) ? True : False
```

5.4.4 Theoretical Analysis

I theoretically prove that my algorithms (**WMGen**, **Mark**, **Verify**) form a qualified watermarking scheme for NAS architectures. I first assume the search space restricted by the watermark is still large enough for the owner to find a qualified architecture.

Assumption 5.1. Let $\mathbb{S}_0$, $\mathbb{S}$ be the search spaces before and after restricting a watermark in a NAS method, $\mathbb{S}_0 \supseteq \mathbb{S}$. $\mathfrak{A}_0 \in \mathbb{S}_0$ is the optimal architecture for an arbitrary data distribution $\mathcal{D}$. $\mathfrak{A}$ is the optimal architecture in $\mathbb{S}$, The model accuracy of $\mathfrak{A}$ is no smaller than that of $\mathfrak{A}_0$ by a relaxation of $\frac{\epsilon}{N}$.

I further assume the existence of an ideal analyzer that can recover the watermark from the given side-channel trace.

Assumption 5.2. Let mk and vk be the marking and verification keys of a DNN architecture $\mathfrak{A} = \{c_1, ..., c_N\}$. For $\forall\ mk, vk$, and $\mathfrak{A}$, there is a leakage analyzer P that is capable of recovering all the stamps of $\{c_i\}_{i=1}^N$ from a corresponding cache side-channel trace.

With the above two assumptions, I prove the following theorem:

Theorem 5.1. *With Assumptions 5.1-5.2, Algorithms 2-4 form a watermarking scheme that satisfies the properties of effectiveness, usability, robustness, and uniqueness in Section 5.3.2.*

5.5 Side Channel Extraction

Given a suspicious model, I aim to extract the embedded watermark using cache side channels. Past works proposed cache side channel attacks to steal DNN models [79, 80]. However, these attacks are only designed for conventional DNN models and cannot extract NAS models with more sophisticated operations. Besides, the adversary needs to have the knowledge of the target model's architecture family (i.e., the type of each layer), which cannot be obtained in my case.

I design an improved methodology over Cache Telepathy [80] to extract the architecture of NAS models by monitoring the side-channel pattern from the BLAS

library, which actually have been well introduced in Chapter 3. In this section, I provide a further detailed analysis about the leakage pattern of common operations used in NAS, and describe how to identify the operation type and hyper-parameters.

5.5.1 Method Overview

State-of-the-art NAS algorithms [129, 132, 133, 243] commonly adopt eight classes of operations: (1) identity, (2) fully connected layer, (3) normal convolution, (4) dilated convolution, (5) separable convolution, (6) dilated-separable convolution, (7) pooling and (8) various activation functions. Note that although *zeroize* is also a common operation in NAS, I do not consider it, as it just indicates a lack of connection between two nodes and is not actually used in the search process.

As shown in Chapter 3, I take the *itcopy* and *oncopy* APIs in OpenBLAS as the monitoring targets. Since these two APIs are used to load matrix data into the cache, I can analyze the access pattern to them to reveal the dimension information of computing matrix. Besides, the variance of API access pattern also leaks the type of running operation. To remind that, Figure 5.2 illustrates the leakage patterns of four representative operations with a sampling interval of 2000 CPU cycles. Different operations have distinct patterns of side-channel leakage. By observing such patterns, I can identify the type of the operation.

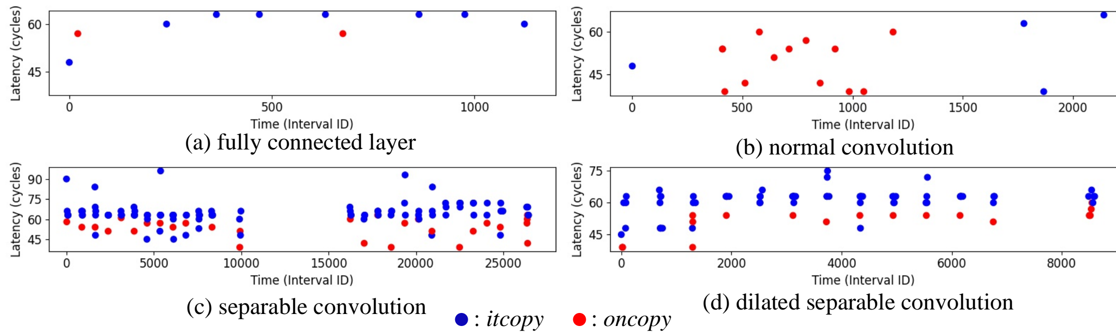


FIGURE 5.2: Event sequences of four representative operations in NAS models.

Finally, I derive the hyper-parameters of each operation based on the inferred matrix dimension. The relationships between the hyper-parameters of various operations and the dimensions of the transformed matrices would be well studied. Below I give detailed descriptions on the recovery of each NAS operation.

5.5.2 Recovery of NAS Operations

Fully connected (FC) layer. This operation can be transformed to the multiplication of a learnable weight matrix θ $(m \times k)$ and an input matrix in $(k \times n)$, to generate the output matrix out $(m \times n)$. m denotes the number of neurons in the layer; k denotes the size of the input vector; and n reveals the batch size of the input vectors. Hence, with the possible values of (m, n, k) derived from the iteration counts of *itcopy* and *oncopy*, hyper-parameters (e.g., neurons number, input size) of the FC layer can be recovered. The number of FC layers in the model can also be recovered by counting the number of matrix multiplications. Figure 5.2(a) shows the pattern of a classifier with two FC layers, where two separate clusters can be easily identified.

Normal convolution. Although this operation was adopted in earlier NAS methods [132, 137], recent works [129, 130, 139] removed it from the search space as it is hardly used in the searched cells. However, given it is the basis of the following complex convolutions, I still perform detailed analysis about it.

Figure 5.3 shows the structure of a normal convolution at the i-th layer (upper part), and how it is transformed to a matrix multiplication (lower part). Each patch in the input tensor is stretched as a row of matrix in_i, and each filter is stretched as a column of matrix F_i. Hence, the number of filters D_{i+1} can be recovered from the column size n of the filter matrix F_i. The kernel size R_i can be revealed from the column size $k = R_i^2 D_i$ of the matrix in_i, as I assume D_i has been obtained from the previous layer. With the recovered R_i, the padding size P_i can be inferred as the difference between the row sizes of out_{i-1} and in_i, which are $W_i \times H_i$ and $(W_i - R_i + P_i + 1)(H_i - R_i + P_i + 1)$, respectively. The stride can be deduced based on the modification between the input size and output size of the convolution. In a NAS model, the convolved feature maps are padded to preserve their spatial resolution, so I have $P_i = R_i - 1$. A normal cell takes a stride of 1, while a reduction cell takes a stride of 2.

In terms of the leakage pattern, a normal convolution is hard to be distinguished from a FC layer, as both of their accesses to the *itcopy* and *oncopy* functions can be denoted as $xI - yO - zI$, where (x, y, z) indicate the repeated times of the functions, determined by the operation hyper-parameters. This is why Cache Telepathy [80]

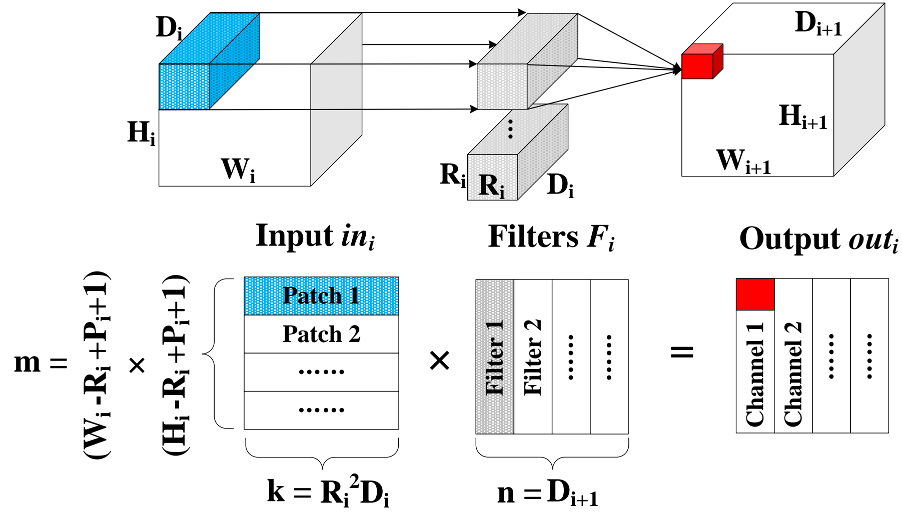


FIGURE 5.3: Implementing a convolution operation as matrix multiplication

needs to know the architecture family of the target DNN to distinguish the operations. Figure 5.2(b) shows the leakage pattern of a normal convolution. In the NAS scenario, since the normal convolution is generally used at the preprocessing stage, while the FC layer is adopted as the classifier at the end, they can be distinguished based on their locations.

Dilated convolution. This operation is a variant of the normal convolution, which inflates the kernel by inserting spaces between each kernel element. I use the dilated space d to denote the number of spaces inserted between two adjacent elements in the kernel. The conversion from the hyper-parameters of a dilated convolution to the matrix dimension is similar with the normal convolution. The only difference is the row size m of the input matrix in_i, i.e., the number of patches. Due to the inserted spaces in the kernel, although the kernel size is still R_i^2, the actual size covered by the dilated kernel becomes $R_i'^2$, where $R_i' = R_i + d(R_i - 1)$. This changes the number of patches to $(W_i - R_i' + P_i + 1)(H_i - R_i' + P_i + 1)$. As a dilated convolution is normally implemented as a dilated separable convolution in practical NAS methods [129, 130], the leakage pattern of the operation will be discussed with the dilated separable convolution.

Separable convolution. According to [80], the number of consecutive matrix multiplications with the same pattern reveals the batch number of a normal convolution. However, I find this does not hold in the separable convolution, or precisely, the depth-wise separable convolution used in NAS. This is because the separable convolution decomposes a convolution into multiple separate operations, which

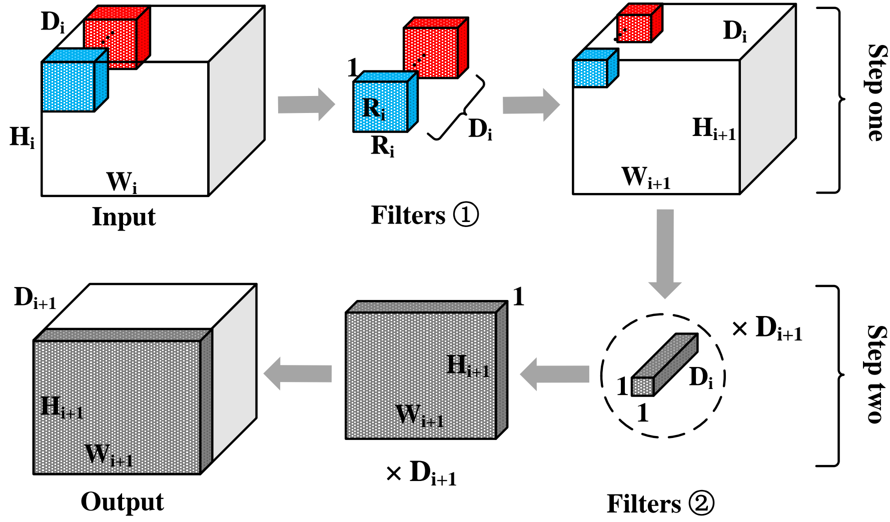


FIGURE 5.4: Procedure of separable convolutions.

can incur the same conclusion that the number of the same patterns equals to the number of input channels.

A separable convolution aims to achieve more efficient computation with less complexity by separating the filters. Figure 5.4 shows a two-step procedure of a separable convolution. The first step uses D_i filters (Filters ①) to transform the input to an intermediate tensor, where each filter only convolves one input channel to generate one output channel. It can be regarded as D_i normal convolutions, with the input channel size of 1 and the filter size of $R_i^2 \times 1$. These computations are further transformed to D_i consecutive matrix multiplications with the same pattern, which is similar as a normal convolution with the batch size of D_i. But the separable convolution has much shorter intervals between two matrix multiplications, as they are parts of the whole convolution, rather than independent operations. In the second step, a normal convolution with D_{i+1} filters (Filters ②) of size $1^2 \times D_i$ is applied to the intermediate tensor to generate the final output.

In summary, the leakage pattern of the separable convolution is fairly distinguishable, which contains D_i consecutive clusters and one individual cluster at the end. Note that in a NAS model, the separable convolution is always applied twice [129, 130, 132, 137, 248] to improve the performance, which makes its leakage pattern more recognizable. Figure 5.2(c) shows the trace of a separable convolution. There are clearly two parts following the same pattern, corresponding to

the two occurrences of the operation. Each part contains 12 consecutive same-pattern clusters to reveal $D_i = 12$, and an individual cluster denoting the last 1×1 convolution.

Dilated separable (DS) convolution. This operation is the practical implementation of a dilated convolution in NAS. The DS convolution only introduces a new variable, the dilated space d, from the separable convolution. Hence, this operation has similar matrix transformation and leakage pattern as the separable convolution, except for two differences. First, R_i is changed to $R'_i = R_i + d(R_i - 1)$ in calculating the number of patches $m = (W_i - R_i + P_i + 1)(H_i - R_i + P_i + 1)$ in Step One. Second, a DS convolution needs much shorter execution time. Figure 5.2(d) shows the leakage pattern of a DS convolution with the same hyper-parameters as a separable convolution depicted in Figure 5.2(c), except that the dilated space $d = 1$. It is easy to see the performance advantage of the DS convolution (8400 intervals) over the separable convolution (10000 intervals) under the same configurations. The reason is that the input matrix in a DS convolution contains more padding zeroes to reduce the computation complexity. Besides, the DS convolution does not need to be performed twice, which also helps us distinguish it from a separable one.

Skip connect. The operation is also called *identity* in the NAS search space, which just sends out_i to in_j without any processing. This operation cannot be directly detected from the side-channel leakage trace, as it does not invoke any GEMM computations. While [80] argues the skip can be identified as it causes a longer latency due to the introduction of an extra merge operation, it is not feasible in a NAS model. This is because in a cell, each node has an add operation of two inputs and the skip operation does not invoke any extra operations. So there is no obvious difference between the latency of skip and the normal inter-GEMM intervals.

Pooling. I assume the width and height of the pooling operation is the same, which is default in all practical implementations. Given that pooling can reduce the size of the input matrix in_i from the last output matrix out_{i-1}, the size of the pooling layer can be obtained by performing square root over the quotient of the number of rows in out_{i-1} and in_i. In general, pooling and non-unit striding cannot be distinguished as they both reduce the matrix size. However, in a NAS model, non-unit striding is only used in reduction cells which can double the channels. This

information can be used for identification. Pooling cannot be directly detected as it does not invoke any matrix multiplications in GEMM. But it can introduce much longer latency (nearly 1.5× of the normal inter-GEMM latency) for other computations. Hence, I can identify this operation by monitoring the matrix size and execution intervals. While monitoring the BLAS library can only tell the existence of the pooling operation, the type can be revealed by monitoring the corresponding pooling functions in the deep learning framework.

Other DNN components. Besides the above operations, other common components like batch normalization, dropout and activation functions are also critical to the model performance. My method can be generalized to watermark these components as well, by just changing the monitored library targets. For instance, to protect activation functions, e.g., *relu* and *tanh*, I can monitor accesses to the corresponding function APIs in Pytorch.

5.6 Evaluation

5.6.1 Experimental Setup

NAS implementation. My scheme is independent of the search strategy, and can be applied to all cell-based NAS methods. I mainly focus on the CNN tasks, and select a state-of-the-art NAS method GDAS [130], which can produce qualified network designs within five GPU hours. I follow the default configurations to perform NAS [130, 132]: the search space of a CNN cell contains: *identity (skip), 3×3 and 5×5 separable convolutions (SC), 3×3 and 5×5 dilated separable convolutions (DS), 3×3 average pooling (AP), 3×3 max pooling (MP)*. The discovered cells are then stacked to construct DNN models. I adopt CIFAR10 as the proxy dataset to search the architecture, and train CNN models over different datasets, e.g., CIFAR10, CIFAR100, ImageNet.

Side channel extraction. To capture the side-channel leakage of CNN models, I monitor the *itcopy* and *oncopy* functions in OpenBLAS. I adopt the FLUSH+RELOAD side-channel technique [78], but other methods can achieve my goal as well. I inspect the cache lines storing these functions at a granularity of 2000 CPU cycles to obtain accurate information.

5.6.2 Effectiveness

5.6.2.1 Key Generation

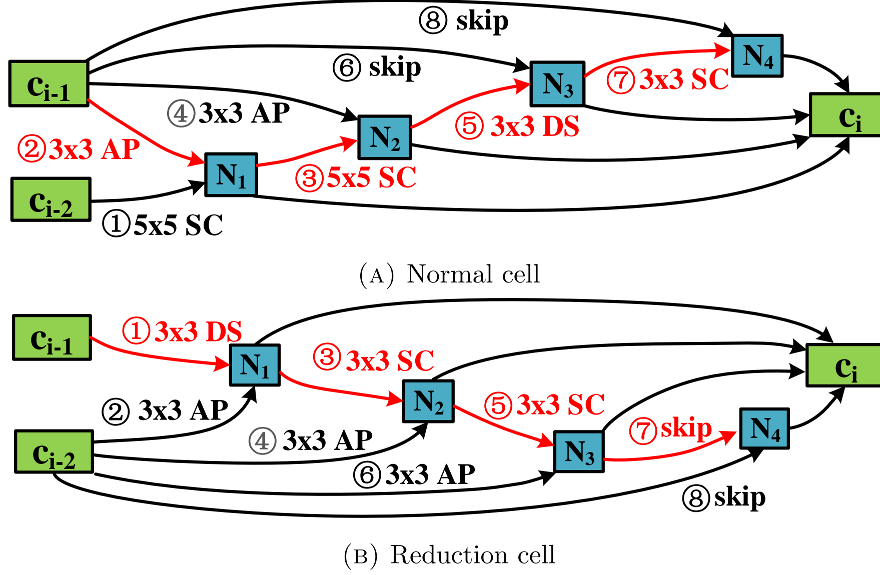
A NAS method generally considers two types of cells. So I set the same stamp for each type. Then the marking key can be denoted as $mk = (mk_n, mk_r)$, where $mk_n = \{s_{en} : s_{on}\}$ and $mk_r = \{s_{er} : s_{or}\}$ represent the stamps embedded to the normal and reduction cells, respectively. Each cell has four computation nodes ($\mathcal{B} = 4$), and I set the number of stamp edges $n_s = 4$ for both cells, indicating four causal edges in each cell are fixed and attached with random operations. I follow Algorithm 2 to generate one example of mk (Table 5.1). The verification key $vk = (vk_n, vk_r)$ is also recorded, where $vk_n = s_{on}$ and $vk_r = s_{or}$.

mk_n	s_{en}	$c_{i-2} \to \mathcal{N}_0$	$\mathcal{N}_0 \to \mathcal{N}_1$	$\mathcal{N}_1 \to \mathcal{N}_2$	$\mathcal{N}_2 \to \mathcal{N}_3$
	s_{on}	3×3 AP	5×5 SC	3×3 DS	3×3 SC
mk_r	s_{er}	$c_{i-1} \to \mathcal{N}_0$	$\mathcal{N}_0 \to \mathcal{N}_1$	$\mathcal{N}_1 \to \mathcal{N}_2$	$\mathcal{N}_2 \to \mathcal{N}_3$
	s_{or}	3×3 DS	3×3 SC	3×3 SC	skip

Table 5.1: An example of the marking key mk.

5.6.2.2 Watermark Embedding

I follow Algorithm 3 to embed the watermark determined by mk to the DNN architecture during the search process. Figure 5.5 shows the architectures of two cells searched by GDAS, where stamps are marked as red edges, and the computing order of each operation is annotated with numbers. These two cells are further stacked to construct a complete DNN architecture, including three normal blocks (each contains six normal cells) connected by two reduction cells. The preprocessing layer is a normal convolution that extends the number of channels from 3 to 33. The number of filters is doubled in the reduction cells, and the channel sizes (i.e., filter number) of three normal blocks are set as 33, 66 and 132. I train the searched architecture over CIFAR10 for 300 epochs to achieve a 3.52% error rate on the validation dataset. This is just slightly higher than the baseline (3.32%), where all connections participate in the search process. This shows the usability of my watermarking scheme.



(A) Normal cell

(B) Reduction cell

FIGURE 5.5: Architectures of the searched cells. c_{i-1} and c_{i-2} are the inputs from the previous cells.

5.6.2.3 Watermark Extraction and Verification

Given a suspicious model, I launch a spy process to monitor the activities in OpenBLAS during inference, and collect the side-channel trace. I conduct the following steps to analyze this trace.

First, I check whether the pattern of the whole trace matches the macro-architecture of a NAS model, i.e., the trace has three blocks, each of which contains six similar leakage windows, and divided by two different leakage windows.

Second, I focus on the internal structure of each cell and check if it contains the fixed operation sequence given by vk. Here I only demonstrate the pattern of the first leakage window (i.e., the first normal cell) as an example (Figure 5.6). Other cells can be analyzed in the same way. Recall that in the figure the blue node denotes an access to itcopy and red node denotes an access to oncopy. From this figure, I can observe four large clusters, which can be easily identified according to their leakage patterns that ① , ③ and ⑦ are SCs while ⑤ is a DS. Figure 5.7a shows the measured execution time of these four GEMM operations. An interesting observation is that 5×5 convolution takes much longer time than 3×3 convolution, because it computes on a larger matrix. Such timing difference enables us to identify the kernel size when the search space is limited. Besides, I can also infer that the channel size is 33, since each operation contains $C = 33$ consecutive

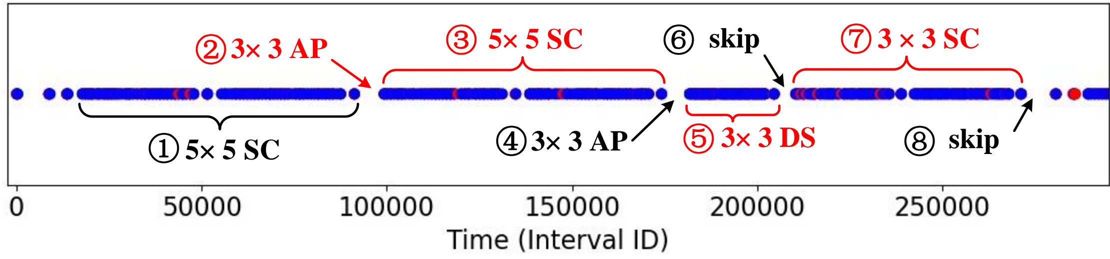
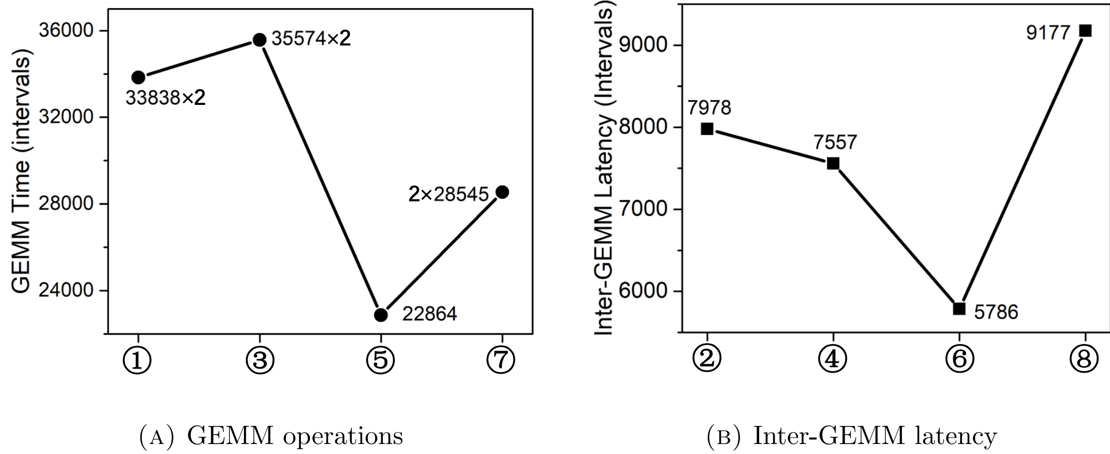


FIGURE 5.6: A side-channel trace of the first normal cell.



(A) GEMM operations (B) Inter-GEMM latency

FIGURE 5.7: Execution time of the operations in a cell

sub-clusters[1]. Figure 5.7b gives the inter-GEMM latency in the cell. The latency of ② and ④ is much larger, indicating they are pooling operations. Particularly, the latency of ⑧ contains two parts: *skip* and interval between two cells. The three small clusters at the beginning of the trace are identified as three normal convolutions used for preprocessing the input. Finally, after identifying the fixed operation sequences (s_o) in all cells, I can claim the architecture ownership of the DNN model.

The above analysis can already give us fair verification results. To be more confident, I further recover the remaining hyper-parameters (in particular, the kernel size) based on their matrix dimensions (m, n, k). Figure 5.8 shows the values of (m, n, k) extracted from $iter_n$ in the normal cell, where each operation contains two types of normal convolutions. For certain matrix dimensions that cannot be extracted precisely, I empirically deduce their values based on the constraints of NAS models. For instance, m is detected to be between [961, 1280]. I can fix it as $m = 1024$ since it denotes the size of input to the cell and 32×32 is the most common setting. The value of n can be easily deduced as it equals the channel

[1]The value of C can be identified if I zoom in Figure 5.6.

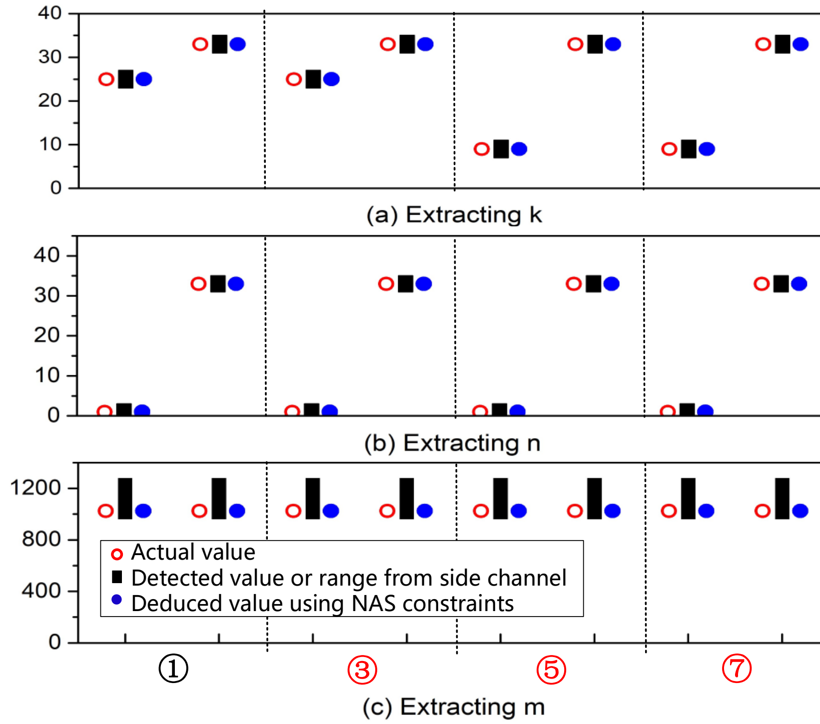


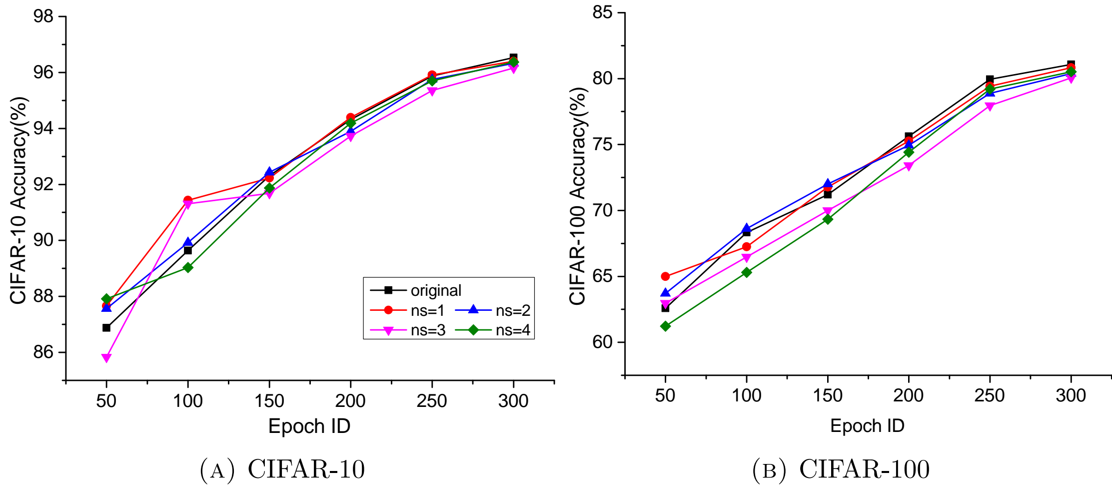
FIGURE 5.8: Extracted values of the matrix parameters (m, n, k).

size. Deduction of k is more difficult, since the filter size k in a NAS model is normally smaller than the GEMM constant in OpenBLAS, it does not leak useful messages in the side-trace trace. However, an interesting observation is that 5×5 convolution takes much longer time than 3×3 convolution, because it computes on a larger matrix. Such timing difference enables us to identify the kernel size R_i when the search space is limited. Analysis on the reduction cells is similar.

5.6.3 Usability

To evaluate the usability property, I vary the number of stamp edges n_s from 1 to 4 to search watermarked architectures. Then I train the models over CIFAR10, CIFAR100 and ImageNet, and measure the validation accuracy. Figure 5.9 shows the average results on CIFAR dataset of five experiments versus the training epochs.

I observe that models with different stamp sizes have quite distinct performance at epoch 100. Then they gradually converge along with the training process, and finally reach a similar accuracy at epoch 300. For CIFAR10, the accuracy of the original model is 96.53%, while the watermarked model with the worst performance



(A) CIFAR-10 (B) CIFAR-100

FIGURE 5.9: Top-1 validation accuracy

(n_s = 3) gives an accuracy of 96.16%. Similarly for CIFAR100, the baseline accuracy and worst accuracy (n_s = 4) are 81.07% and 80.35%. I also check this property on ImageNet. Since training an ImageNet model is quite time-consuming (about 12 GPU days), I only measure the accuracies of the original model and two watermarked models (n_s = 2 and 4), which are also roughly the same (73.97%, 73.16% and 72.73%). This confirms my watermarking scheme does not affect the usability of the model.

Selection of the stamp size. The setting of the stamp size is a trade-off between model usability and watermark reliability. Since my watermark scheme requires the stamp edges to form a dependent path to be robust against operation shuffling attacks (Section 5.6.4), the largest number of stamps is restricted by the number of nodes in the NAS cell. For conventional NAS architectures, the range of stamp size is [0, 4]. My evaluation results (Figure 5.9) indicates that 4 stamp edges incur negligible performance degradation. Therefore, I recommend to adopt this setting in my watermark scheme.

5.6.4 Robustness

I consider the robustness of my watermarking scheme against four types of scenarios.

System noise. It is worth noting that the noise in the side-channel traces (e.g., from the system activities, interference with other applications) could possibly

make it difficult for the model owner to identify the watermarks. To evaluate this, I follow previous works [83] to inject up to 30% scales of Gaussian noise into the time interval between events in the side-channel trace, which can well simulate the system noise. I find that it is still feasible to extract the watermarking operations with high fidelity. I conclude that the impact of system noise on operation extraction is actually negligible. The reason behind is that the most important operation features, such as the operation class, channel size and kernel size, are all revealed by analyzing the holistic pattern of side-channel leakage traces. System noise that just disturbs local patterns will not mislead the inference of these operation features. The recovery of matrix dimensions (m, n, k) is indeed affected by side-channel noise, but as I only need to deduce a range of these parameters, such impact is acceptable. Besides, according to my threat model in Section 3.3.1, the model owner takes control of the host TEE platform, so he can disable other applications on the same machine to further improve the verification reliability.

Model transformation. Prior parameter-based solutions [233–235] are proven to be vulnerable against model fine-tuning or image transformations [237–240]. In contrast, my scheme is robust against these transformations as it only modifies the network architecture. First, I consider four types of fine-tuning operations evaluated in [233] (*Fine-Tune Last Layer*, *Fine-Tune All Layers*, *Re-Train Last Layer*, *Re-Train All Layers*). I verify that they do not corrupt my watermarks embedded to the model architecture. Second, I consider model compression. Common pruning techniques set certain parameters to 0 to shrink the network size. The GEMM computations are still performed over pruned parameters, which give similar side-channel patterns. Figures 5.10(a)-(c) show the extraction trace of the first normal cell after the entire model is pruned with different rates (0.3, 0.6, 0.9) using L_2-norm. Figure 5.10(d) shows one case where I prune all the parameters in the first normal cell. I observe that a bigger pruning rate can decrease the length of the leakage window, as there are more zero weights to simplify the computation. However, the pattern of the operations in the cell keeps unchanged, indicating the weight pruning cannot remove the embedded watermark.

Model obfuscation. An adversary may also obfuscate the inference execution to interfere with the verification results. (1) He can shuffle the orders of some operations which can be executed in parallel. However, since the selected stamp operations are in a path, they have high dependency and must be executed in the

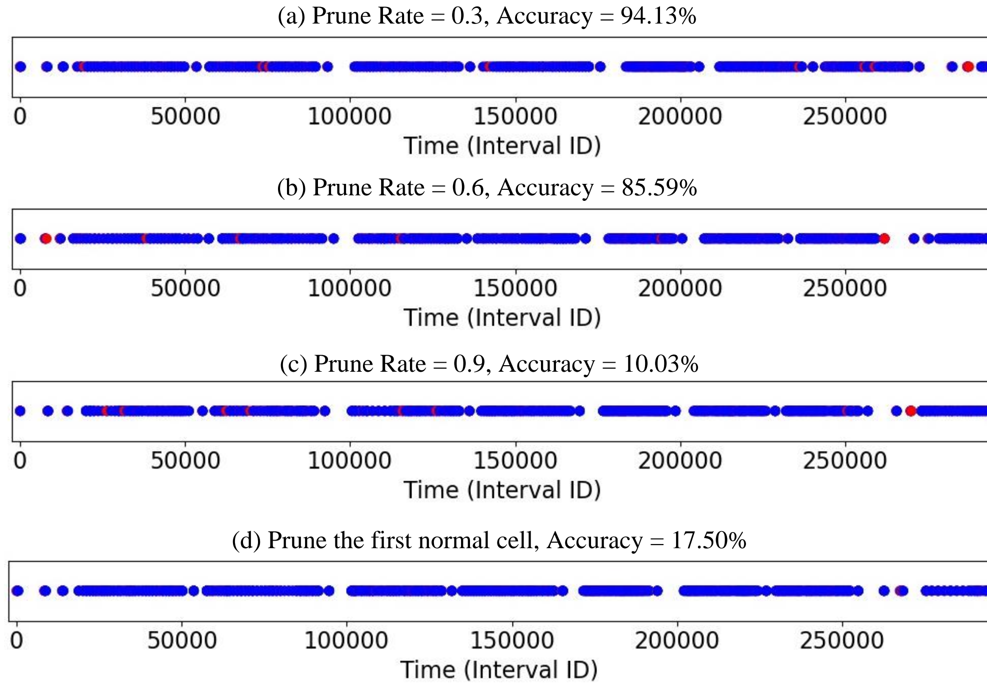


FIGURE 5.10: Side-channel traces of weighting pruned models.

correct order. Hence, I can still identify the fixed operation sequence from the leakage trace of obfuscated models. (2) The adversary can add useless computations (e.g., matrix multiplications), operations or neurons to obfuscate the side-channel trace. Again, the critical stamp operations are still in the trace, and the owner is able to verify the ownership regardless of the extra operations. (3) The adversary may add useless cell windows to obfuscate the watermark verification.

Figure 5.11 illustrates the leakage pattern of the original cell as well as the cells after being obfuscated by above two techniques. Specifically, in Figure 5.11(b), the attacker shuffles the operation execution order, which first executes ②, ④, ⑥ and ⑧ and then runs the watermarked path. I can see that the watermark (i.e., fixed operations) can still be identified in the sequence. In Figure 5.11(c), the attacker adds an unused 3×3 separable convolution (red block) in the pipeline, which does not affect the watermark extraction, as the fixed sequence of stamp operations remains. In short, *the stamp operations must be executed sequentially and cannot be removed in a lightweight manner.* This makes it difficult to remove the watermarks in the architecture.

Figure 5.12 shows the influence of injected useless cell windows. In the side-channel trace, it contains three cell windows, where ① and ③ are NAS cell windows and

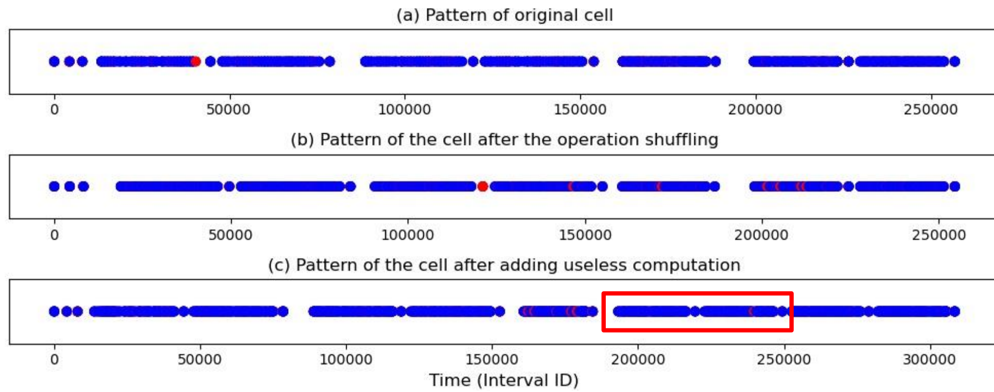


FIGURE 5.11: Traces of obfuscated models.

② is the injected useless cell window. I just need to check if the monitored side-channel trace contains N identical cells and identify if the watermark exists in the cells. Even there are other cells, I can also claim that this model is watermarked and then require for further arbitration.

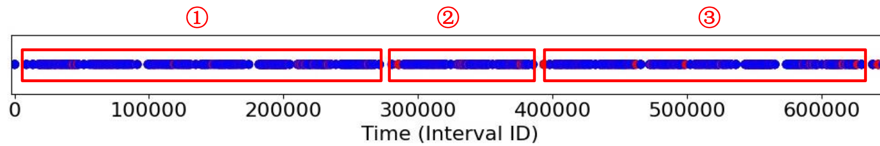


FIGURE 5.12: Influence of useless cell windows.

Structure pruning. I further consider the structured pruning, which can explicitly modify the model structure. This technique is indeed possible to remove my watermark embedded into the network architecture. However, it has two drawbacks: (1) since the watermarking key is secret, the adversary does not know which operation should be pruned; (2) Pruning the stamp operation can cause significant performance drop. To validate this, I random prune 1 to 4 stamp operations in the normal cell, and Table 5.2 shows the prediction accuracy of pruned models on CIFAR10. For the case of pruning one stamp operation, I give four prediction accuracy values corresponding to four possible pruning scheme (pruning one operation from ② , ③ , ⑤ and ⑦ in Fig. 5.5a). For models with more than one stamp operations pruned, I give the average accuracy of pruned models. I observe that even only pruning one stamp operation can lead to great accuracy drop (96.53% to 55.62%). Hence, removing the watermark with structured pruning is not practical.

Note that an adversary can leverage some powerful methods (e.g., knowledge distillation [249, 250]) to fundamentally change the architecture of the target model and possibly erase the watermarks. However, this is not flagged as copyright violation,

# of pruned stamp ops	0	1	2	3	4
Accuracy (%)	96.53	89.34/93.38/ 78.71/55.62	54.89	44.52	37.66

TABLE 5.2: Accuracy of structured pruned models on CIFAR10

since the adversary needs to spend a quantity of effort and cost (computing resources, time, dataset) to obtain a new model. This model is significantly different from the original one, and is regarded as the adversary's legitimate property.

Parameter binarization. This technique [251] is used to accelerate the model execution by binarizing the model parameters. If corresponding Binary Neural Network (BNN) still adopts the BLAS library to accelerate the matrix multiplications, the side-channel leakage pattern keeps similar. Only the time interval between each monitored API access becomes shorter, as the parameter binarization would cause much faster model execution. Figure 5.13 shows the comparison of side-channel traces between the original NAS cell and binarized cell. I observe that although parameter binarization achieves about 20 times faster inference (2.8e5 vs. 1.4e4 intervals), the leakage trace still keeps the similar pattern. Hence, my scheme can still be applied to verify BNN models. If the BNN model adopts other acceleration libraries, I can also switch to monitor that library to perform similar analysis.

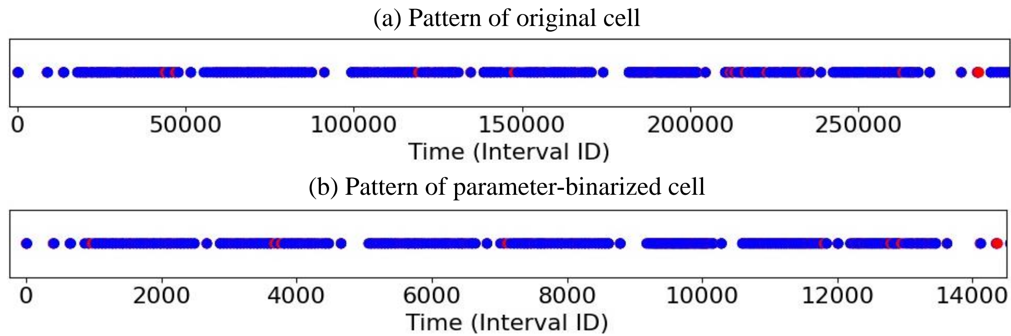


FIGURE 5.13: Influence of parameter binarization.

5.6.5 Uniqueness

Given a watermarked model, I expect that benign users have very low probability to obtain the same architecture following the original NAS method. This is to guarantee small false positives of watermark verification.

The theoretical analysis assumes each edge selects various operations with equal probability. Given a watermarked model, I expect that benign users have a very

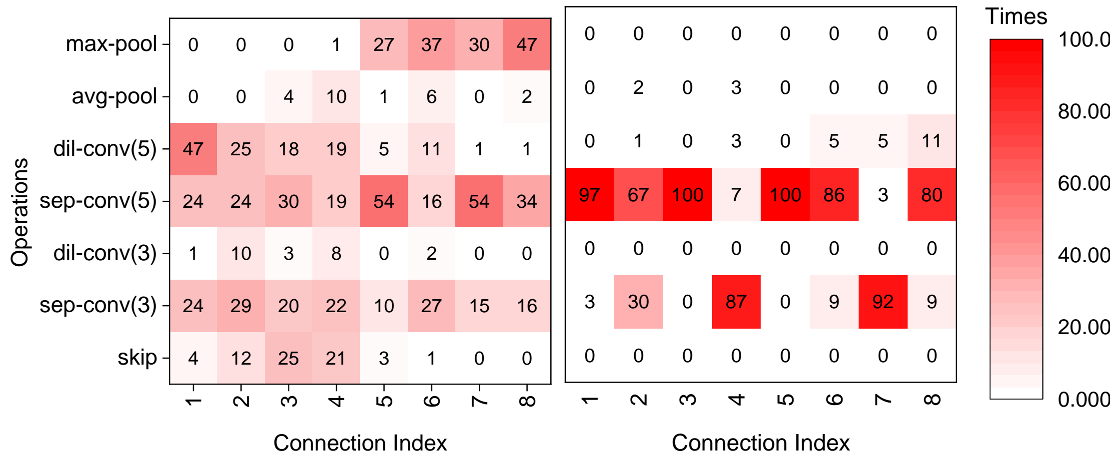


FIGURE 5.14: Operation distributions for a normal cell (left) and reduction cell (right). The connection index is the index of the connection edge in the NAS cell.

low probability to obtain the same architecture following the original NAS method. Without loss of generality, I assume the NAS algorithm can search the same architecture if the search spaces of all cells are the same. Thus, the uniqueness of the watermarked model is decided by the probability that the adversary can identify the same search spaces. Because the marking key is secret, the adversary has to guess the edges and the corresponding operations of each stamp if he wants to identify the same search spaces. Assume the selection of candidate operations is independent and identically distributed, the probability that an operation is chosen on an edge is $\frac{1}{|\mathcal{O}|}$. For a DNN model that contains $\mathcal{B}$ computation nodes, there are $2\mathcal{B}$ connection edges, from which I select n_s causal edges. There are $\binom{2\mathcal{B}}{n_s}$ combinations. Hence, the probability of the stamp collision in a cell can be computed as $\binom{2\mathcal{B}}{n_s} \times (\frac{1}{|\mathcal{O}|})^{n_s}$. In my experiment configurations, the collision rate is smaller than 1.7%. Considering both the normal and reduction cells, the collision rate is smaller than $(1.7\%)^2 \approx 0.03\%$, which can be neglected.

I further empirically evaluate the uniqueness of my watermarking scheme. Specifically, I repeat the GDAS method on CIFAR10 for 100 times with different random seeds to generate 100 architecture pairs for the normal and reduction cells. I find my stamps have no collision with these 100 normal models. Figure 5.14 shows the distribution of the operations on eight connection edges in the two cells. I observe that most edges have some preferable operations, and there are some operations never attached to certain edges. This is more obvious in the architecture of the reduction cell. Such feature can help us to select more unique operation sequence as the marking key. Besides, the collision probability is decreased when the stamp

size n_s is larger. A stamp size of 4 with fixed edge-operation selection can already achieve strong uniqueness.

5.7 Conclusion

In this chapter, I propose a new direction for IP protection of DNNs, which can further enhance the security of confidential AI systems. I show a carefully-crafted network architecture can be utilized as the ownership evidence, which exhibits stronger resilience against model transformations than previous solutions. I leverage Neural Architecture Search to produce the watermarked architecture, and cache side channels to extract the black-box models for ownership verification. Evaluations indicate my scheme can provide great effectiveness, usability, robustness, and uniqueness, making it a promising and practical option for IP protection of AI products.

Chapter 6

Enabling Fast and Secure Function Cold Starts in Confidential Serverless Systems

Confidential serverless systems integrate Trusted Execution Environment with serverless computing, effectively protecting the confidentiality of guest users' functions and data, but also leading to a significant startup latency of function executors. To address this deficiency, I introduce `Neuralyzer`, a new feature layer integrated into confidential serverless systems based on AMD SEV. It is designed to accelerate the extensive startup process of confidential executors, aiming to retain performance improvement offered by serverless computing while ensuring robust security for guest users. Specifically, the `Neuralyzer` layer acts as an isolated and privileged controller within the encrypted function executor, consisting of two key modules: (1) *Restore Module* restoring secure environment with a clean snapshot; (2) *Attestation Module* helping remote users to verify the restored executor state. I mainly focus on serverless systems using unikernel to provide an initial blueprint for analysis. My comprehensive experiments on three baselines and six real-world serverless functions show that the `Neuralyzer` layer can dramatically reduce the startup latency by $76\sim 501\times$ compared to native SEV Virtual Machines (VM), achieving an end-to-end service latency of only a few hundred milliseconds.

6.1 Introduction

Serverless computing, also known as Function as a Service (FaaS), has been widely regarded as the next generation of cloud computing. Major cloud platforms have all introduced their serverless computing services, such as Amazon AWS Lambda [252], Microsoft Azure Functions [253], Google Cloud Functions [254], etc. With its increasing popularity, concerns about confidentiality on such FaaS platforms are also being raised [23, 255]. Given the cloud provider owns the highest privilege of the entire system, he may be benign but curious to directly expose user secrets from the running function executors, e.g., virtual machines or containers. To address this concern, Trusted Execution Environment (TEE) is integrated to achieve *confidential serverless computing* by encapsulating the target function within an encrypted executor [256]. It effectively prevents privileged attackers from extracting sensitive user data, as investigated in various previous works [2, 20–22, 25].

However, the integration of TEE brings significant performance overhead, particularly in terms of startup latency. The characterization of real-world serverless workloads [257] reports that 50% of functions execute within less than 1 second, so serverless applications are extremely sensitive to the service latency. While the median startup latency of a serverless executor in the industry is typically only a few hundred milliseconds [258, 259], TEE-based executors usually require a few seconds. For example, the startup of a 256MB SGX enclave takes 7.1 seconds [2], and the SEV VM requires 13.27 seconds (Section 6.3), both of which are considerably longer than the industry requirement.

Some previous works have attempted to address this issue with focusing on process-level TEE (i.e., Intel SGX [4]), including plug-in enclaves [2] and reusable enclaves [25]. However, confidential serverless computing that leverages system-level virtualization-based TEE has never been discussed. Conversely, state-of-the-art industry serverless frameworks, including Firecrakcer [260], gVisor [261], Kata Containers [262] and Cloud Hypervisor [263], concentrate on lightweight virtualization. Furthermore, from the perspective of confidential computing, the pioneer SGX is becoming outdated because of its overly aggressive threat model and complex confidentiality abstraction between the application and OS. While Intel has deprecated SGX in their latest core processors [264], all major processor manufacturers have shifted to support the alternative path, i.e., *confidential VM*, such as AMD

SEV [265], Intel TDX [6] and ARM CCA [157]. Compared with SGX, VM-based TEE provides a better abstraction for supporting unmodified applications and are more compatible with popular virtualization-based cloud ecosystems, making it the mainstream in future confidential computing. In this chapter, I aim to propose the first work optimizing the heavy startup latency of confidential VMs used in confidential serverless computing.

Conventional serverless computing has introduced numerous optimizations to reduce the service startup latency, such as *Save/Restore* [266–269], *Fork* [268, 270, 271] and *Cache* [257, 272, 273]. Unfortunately, all these optimizations cannot be easily transferred to confidential serverless computing. For example, Save/Restore is quite slow for TEE executors due to the additional en(de)cryptions, and Fork is infeasible given each TEE executor is encrypted by a distinct cipher key. Cache is naturally suitable for confidential serverless computing. Given the limitation of available TEE resource and expensive cost of switching TEE environments, the best way to accelerate the service response is keeping the encrypted executor warm and reusing it for subsequent invocations. However, such sharing leads to potential security issues, where the attacker first using the executor may corrupt the environment to steal secrets of subsequent users. I will discuss more details about these solutions in Section 6.3.

To enable the caching strategy for confidential serverless computing, I propose `Neuralyzer`, a new feature layer integrated into confidential executors (i.e., VMs) as an isolated and privileged controller. With this `Neuralyzer` layer locating inside the encrypted executor, I can retain the performance improvement offered by cached executors while ensuring robust security of user secrets. However, the design of `Neuralyzer` is not trivial, which poses three significant questions: (1) *How to eliminate the environment corruptions crafted by malicious user?* (2) *How to prove to subsequent users that the cached executor is clean and secure?* (3) *How to ensure the* `Neuralyzer` *layer itself is not corrupted by malicious user?*

Aiming to overcome the above challenges, the core design of `Neuralyzer` is derived from three insights: (1) *Corrupted executor environment can be recovered with a clean snapshot.* By saving the initial environment as a clean snapshot, I can always recover the executor to a secure state, even malicious users have corrupted the environment. (2) *The recovered executor state can be verified through attestation.* The integrity and confidentiality of the restored executor can be verified by remote

users with specific attestation mechanisms, which provide proof of security. (3) *There exists a new TEE hardware feature allows for further division of privilege levels within an executor.* This feature has been achieved in AMD SEV and Intel TDX, capable of isolating `Neuralyzer` components in a untouchable layer with the highest internal privileges. In this chapter, I mainly focus on the AMD SEV platforms, where the feature is referred to as Virtual Machine Privilege Levels (VMPL).

Incorporating these insights, I build the `Neuralyzer` layer within a confidential SEV VM. It integrates the advantages of caching optimization and confidential computing into serverless computing, creating a fast and secure serverless service platform. `Neuralyzer` operates at the highest privilege layer (VMPL0) and has full access to the entire VM memory space. In contrast, the guest system executing the target serverless function runs at a lower privilege level (VMPL3) and is isolated from the `Neuralyzer` layer. `Neuralyzer` consists of two essential system components: (1) *Restore Module*, responsible for saving a clean snapshot of the guest system and later restoring the snapshot to recover secure executor environment. (2) *Attestation Module*, enabling a nested attestation mechanism for remote users to verify the restored executor states.

In this chapter, I adopt the unikernel as the guest system to provide an initial blueprint for analysis, as it can significantly simplify the snapshot design. More importantly, the adoption of unikernel is also a growing trend in the field of serverless computing [274–277]. The adaptation to general-purpose operating systems will be my future work. To extensively assess the performance of `Neuralyzer`, I compare it with three baselines: native SEV VM, Save/Restore optimization and vanilla snapshot method. Experiment results show that `Neuralyzer` achieves the best performance and reduces the startup latency by 76$\sim$ 501$\times$ with only a slight memory penalty. Besides, to assess the end-to-end performance of `Neuralyzer`, I test six real-world serverless functions from the well-known Functionbench suite [278]. The results indicate that these services can be completed within a few hundred milliseconds, aligning with the industry median.

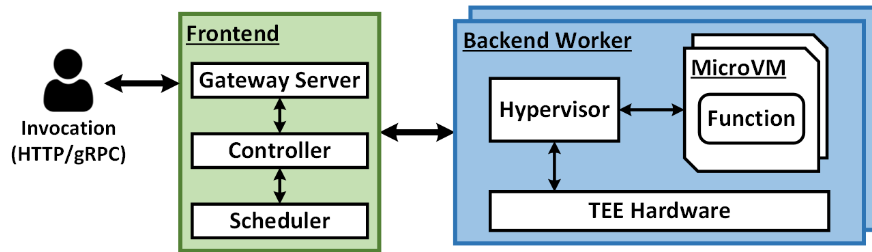


FIGURE 6.1: Architecture of confidential serverless computing

6.2 Background

6.2.1 Confidential Serverless Computing

As a rapidly growing software paradigm for developing and deploying cloud services, serverless computing slices the application functionality into multiple stateless functions. The resource provisioning and scaling for these functions are totally entailed by the cloud provider. However, such complete control also presents an opportunity for a malicious hypervisor to steal or even corrupt user secrets. Following the concept of confidential computing [279], *confidential serverless computing* is proposed to integrate TEE with serverless computing to prevent privileged attackers from compromising the sensitive data in the function executor. Previous works have proposed to protect serverless workloads with Intel SGX [2, 20–22, 25]. With the rising of VM-based TEE, e.g., AMD SEV, more attention is being focused on this emerging alternative. Multiple industry open-source projects have provided the support to SEV-based confidential serverless computing, such as Kata Containers [280] and Podman Libkrun[281]. Figure 6.1 illustrates the typical architecture of the VM-based confidential serverless computing, where the backend function executor (i.e., MicroVM) is encapsulated within the TEE sandbox, and the frontend server is employed to oversee the communication and balance the workload. As most serverless functions are simple enough to complete the execution within 1 second, the significant cold start latency caused by setting up the TEE sandbox environment becomes a critical factor [257, 282].

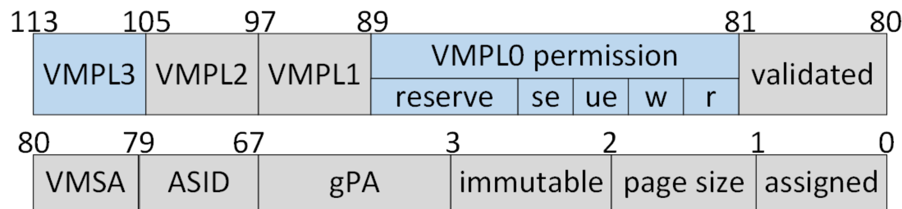


FIGURE 6.2: Fields of an RMP entry [1, Table 15-36]

6.2.2 AMD SEV

Secure Encrypted Virtualization (SEV) is a new hardware extension in AMD processors [265], which encrypts the VM memory and register states to protect user secrets from physical attacks (e.g., cold boot and DMA attacks) and privileged software attacks (e.g., malicious hypervisors). After two initial versions (SEV [5] and SEV-ES [283]), a stable version SEV-SNP [159] was eventually introduced. In this chapter, I directly use SEV to denote the latest SEV-SNP, which further introduces multiple features:

VMSA/GHCB. A new data structure named *VM Save Area (VMSA)* is introduced to automatically save/restore register states when the VM is trapped/resumed. The VMSA page is encrypted from the hypervisor to protect VM register states. However, sometimes the hypervisor needs to access a VM's state to perform emulation. Hence, another data structure called *Guest-Hypervisor Communication Block (GHCB)* is designed to let a VM selectively expose its state to the hypervisor. For example, when an SEV VM executes WRMSR, it only needs to expose EAX, ECX and EDX registers to the hypervisor, as they are operands of that instruction and sufficient for emulation. AMD standardizes the GHCB format for the inter-operation with any supporting hypervisor [284].

RMP/VMPL. To protect the memory integrity, SEV introduces the *Reverse Map Table (RMP)* to establish a one-to-one mapping between the host physical address (hPA) and guest physical address (gPA). Each address mapping corresponds to a RMP entry containing security attributes of this physical page, as shown in Figure 6.2. VMPL stands for *Virtual Machine Privilege Levels*, which provide hardware isolated abstraction layers within a VM for additional security controls [159]. SEV has four VMPLs (0-3) where a smaller number indicates a higher privilege. Each VMPL has its own ring 0-3 and hence its own user and kernel modes. While different VMPLs share the same guest physical memory, they have

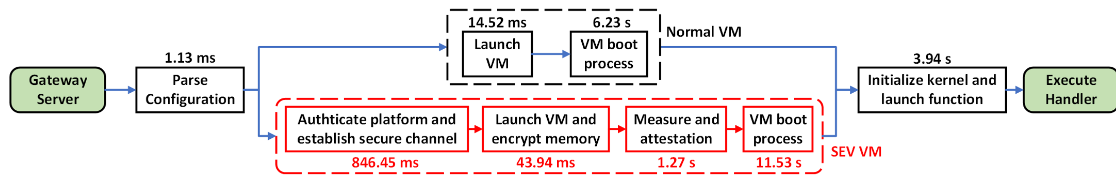


FIGURE 6.3: Boot process of normal VM and SEV VM in serverless computing.

various permission views. VMPL0 is enabled with full permissions: readable (r), writable (w), user executable (ue) and supervisor executable (se), and it can control the permission setting of lower-privileged VMPLs. Each VMPL has its own VMSA, and its associated CPU state is automatically saved/restored by the hardware when switching VMPLs.

6.2.3 Unikernel

The Unikernel is composed of a minimal operating system and a single application, making it well-suited for serverless computing [274–277]. By integrating with a library of OS components, e.g., memory management, scheduler, network stack and device drivers, the application can run as a single process at an elevated privilege level. Finally, the unikernel is built as a standalone binary image that is bootable on (virtual) hardware [285]. The deployment of unikernels brings multiple advantages: (1) The memory footprint of unikernels is much smaller than general OSes [286]. (2) The deployment of simplified unikernels also significantly reduces the cold startup latency (6×-10×) of function executors [287]. (3) Unikernels operate in a single address space, where the kernel functionality can be specialized for some highly-restricted execution domains, like edge devices [288, 289] or SGX enclaves [290, 291]. In this chapter, I also leverage the simplified memory layout of the unikernel to reduce the difficulty of snapshot design.

6.3 Motivation

6.3.1 Startup Latency of SEV VM

Given the high compatibility of SEV with virtualization designs, actually all VM-based serverless designs, like FireCracker [260] and gVisor [261], can be used to

achieve confidential serverless computing by directly replacing the normal VM with SEV VM, as demonstrated in some open-source projects [280, 281]. However, this field is still in its early stage. Due to the ongoing rapid development of SEV support, these downstream projects utilizing SEV are usually immature and fail to incorporate the latest features, e.g., RMP and VMPL. Hence, to accurately identify the bottleneck of SEV executors, I adopt the latest AMD official SEV VM implementation [85] as my analysis target.

On a serverless platform, the first step of invoking a function is to prepare an executor. Figure 6.3 illustrates the boot process of an executor, i.e., a normal VM or SEV VM with memory size of 256MB. To prepare a normal VM, the hypervisor first parses the VM configuration, then launches it from the guest image provided by remote users. After that, the guest kernel is initialized to launch the workload function. From Figure 6.3, I can see that the primary latency stems from the last two steps, which totally takes about 10 seconds.

However, with the integration of SEV VM, the boot process has changed significantly. First, the remote user needs to authenticate the SEV platform based on the certificate chain provided by the platform owner, and establish a secure communication channel to the SEV hardware. Then the hypervisor launches the VM and calls the SEV hardware to encrypt the VM memory. After that, the SEV hardware measures the initial plaintext VM memory and calculates a hash, which is protected by the secure channel and sent to the remote user together with other related information, such as the SEV API version and guest policy. The hash would be attested by the user to verify the VM state. While older SEV versions only support attestation during guest launch, the latest SEV-SNP has introduced a more flexible attestation mechanism that can be employed at any time [159]. Meanwhile, the VM will be further booted, which consumes much longer time (11.53s) than the normal VM, as it contains multiple additional operations for SEV features, like enabling the SEV guest driver and configuring SEV options.

Figure 6.4 shows the startup latency of the executor with various guest memory sizes, in which the workload function is a simple helloworld program. From the figure, I can see that the startup latency of SEV VM keeps stable with the increased memory size, while that of SGX enclave increases remarkably. Such overhead in SGX actually comes from the memory contention caused by the `EADD` instruction. In contrast, SEV VM directly reuses the existing memory allocation mechanism,

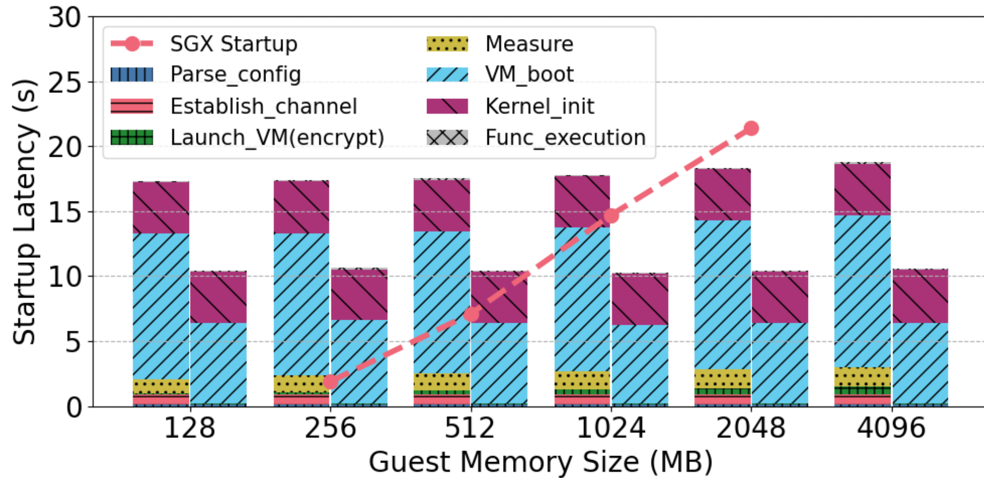


FIGURE 6.4: Startup latency of the SEV VM (left bars) and the normal VM (right bars), and SGX (line, sourced from [2]).

thereby avoiding the performance drop caused by increased guest memory. Furthermore, the most crucial observation from the figure is that the function execution time (∼1ms) only accounts a negligible part in the overall startup latency. This is a catastrophic waste of resources, as the time spent on meaningful computation is less than 1‱. This leads to my motivation that *the initialization stages before the function execution should be bypassed.*

6.3.2 Analysis of Startup Optimizations

In fact, the issue of startup latency has consistently been the focal point in serverless computing. There have been numerous efforts aiming to mitigate this issue, which can be mainly classified into three categories. Unfortunately, these existing optimizations cannot be trivially migrated to confidential computing scenarios. In the following, I assess them from three perspectives: feasibility, performance and security, as shown in Table 6.1.

Cache [257, 272, 273]. By launching executors in advance or caching finished executors, future functions can directly reuse cached ones with nearly no startup latency. However, keeping idle executors alive occupies memory resources, where the overhead can reach even hundreds of GBs [260]. Figure 6.5 shows the performance of six real-world serverless functions (Table 6.2), running on a cached normal VM with the memory size of 256MB. I can see that all functions can be

Feature	Cache	Fork	Save/Restore	Neuralyzer
Startup Speed	Very Fast	Fast	Medium	Very Fast
Resource Usage	High	Medium	Small	High
Evaluate in confidential serverless computing				
Feasibility	✓	×	✓	✓
Performance	✓	♦✓	×	✓
Security	×	♦✓	✓	✓

TABLE 6.1: Evaluations of existing optimizations. ♦✓ denotes the feature is enabled if future TEE hardware supports Fork.

invoked within about 1ms, and the memory footprints is the same as the allocated memory size. The memory overhead for n cached executors can be computed as $\mathcal{O}(n)$ [292].

The Cache method is naturally suitable for TEE executors, as the limited TEE resources and high launch overhead make it extremely expensive to perform any environment switch. The performance penalty of following the Save/Restore method is a good example (Figure 6.6). Consequently, keeping the TEE executor warm and caching it for different users seem to be the best solution. However, it poses serious security issues, where malicious users first using the executor can corrupt the environment to expose the secrets of subsequent users. This is why the community advises to discard the TEE executor after each execution [293] and not to adopt the Cache method for cold start optimization.

Fork [268, 270, 271]. This method is improved based on Cache, where a cached executor calls the fork system call to start a new executor. Hence, it only needs one *template* to perform in-memory copy for future invocations. However, such method is infeasible for TEE executors, which prohibit the in-memory copy. Given each TEE executor is encrypted with a distinct cipher key managed by the trusted hardware, forking a TEE executor only copies its encrypted memory contents to another location, which has not been registered in the TEE hardware. Consequently, this so-called new executor is beyond the knowledge of TEE hardware, and cannot be executed without the assistance of trusted hardware. Modifying the TEE hardware design could enable this optimization, which has been studied in SGX [2]: two new instructions are introduced to achieve the plug-in enclave, a variant of the Fork method. In such scenario, the enabled Fork method can provide good performance, even slightly worse than Cache. It can also ensure the security of new executors, as long as the cached template is well protected.

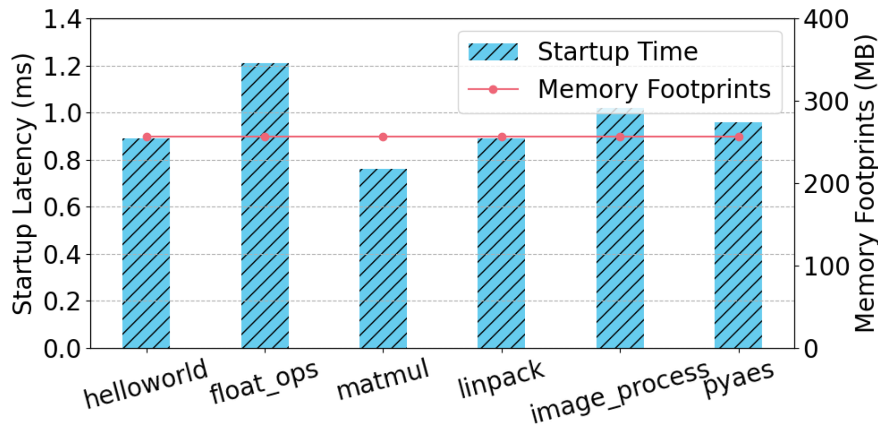


FIGURE 6.5: Caching performance of serverless functions.

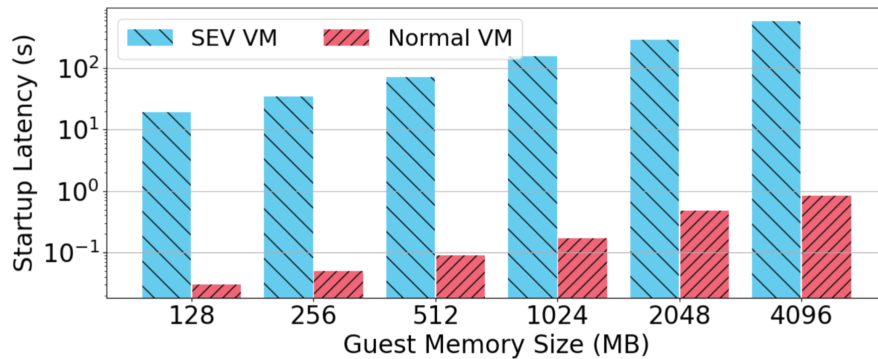


FIGURE 6.6: Performance of Save/Restore method.

Save/Restore [266–269]. This method saves the state of an executor as a specific *snapshot* file. When the same function is invoked, the snapshot is used to restore the checkpoint. It is optimal in resource usage, as it only needs to store one file. However, such method is orders of magnitude slower than the previous two techniques, which becomes even more apparent in the confidential computing. For example, SEV has offered official APIs `SEND_*` and `RECEIVE_*` to achieve snapshot or migration functionality [294]. However, since the snapshot is stored outside the TEE executor, multiple additional en(de)cryptions and environment switches are required to ensure the security, making the method quite slow. Previous works have investigated these APIs, showing their speed is only from 800 KB/s to 5MB/s [215, 295]. In my CPU module, it is slightly faster to reach nearly 7MB/s. Figure 6.6 shows the startup latency of the VM instance when the save/restore method is applied. I observe that restoring the snapshot into SEV VM needs to take minutes or even hours, which is impractical for the serverless computing scenario.

My method. Based on above analysis and evaluation, I intend to design a new confidential serverless system that is compatible with Cache optimization. The goal is to *retain the performance improvement offered by cached executors while ensuring the security of guest users*. Inspired by the insights mentioned in Section 6.1, I try to implement an isolated and privileged `Neuralyzer` layer inside the SEV VM, which can restore the guest system to a verifiable clean state before each new function invocation.

6.4 System Overview

6.4.1 Threat Model

My system design centers around VM-based confidential serverless computing, using SEV VM as the function executor. The SEV VM is cached in the host memory and reused by various users. Following the typical threat model of confidential computing, I identify the privileged hypervisor as a potential attacker with the intent to reveal user secrets. The attacker can exploit any exposed interfaces of the VM and leverage OS capabilities, such as triggering interrupts or invoking VMEXIT. Furthermore, I also consider other users who have previously invoked this cached VM are untrusted. Given that they own the highest privileges of the guest OS, they can corrupt the guest system environment, e.g., manipulating the critical parameters or even injecting persistent backdoors, which threaten the security of subsequent users. The above two attackers can also collaborate to obtain the capabilities from both sides.

I assume the SEV hardware will perform correctly according to its architectural specification. The guest OS within the VM is well-written but could be compromised by malicious users. The initial booting of the SEV VM is deemed trustworthy, as the process integrity has been verified through the hardware attestation. DDoS attacks [296] and attacks against the TEE hardware (e.g., side channels [76, 297, 298], memory extraction [92, 299], voltage glitch [215]) are considered to be out of my scope. Actually, most of these attacks have been well mitigated in the latest SEV version.

6.4.2 Design Principles & Challenges

For practical adoptions, `Neuralyzer` follows three design principles: (1) *Confidential and Secure.* The user secrets should be well protected from the untrusted external hypervisor and internal users who corrupted the guest OS. (2) *Fast and Lightweight.* To fulfill the demands of serverless computing, the cached function executor should be able to respond quickly to user requests while minimizing the memory overhead caused by caching. (3) *Verifiable and Attestable.* The state of guest system should be verifiable to the remote user, who only sends the secret data after the successful attestation.

However, achieving above principles is not trivial, which poses multiple significant challenges:

(1) Protection of layer components. Given that the attacker owns the highest privileges of the guest system, I need to isolate the `Neuralyzer` layer in a untouchable environment to prevent potential corruptions, so that it can operate correctly. Hence, I need to find a secure and isolated area to seal sensitive components of `Neuralyzer`, without incurring significant performance overhead.

(2) Design of snapshot image. To achieve fast response and lightweight deployment, the snapshot saved for recovering clean guest environment cannot be too large. For example, the simplest snapshot design is to directly save the entire memory space of the guest system. However, it would double the memory footprints and slow down the restoring. So I need to minimize the snapshot by identifying which memory portions in the guest system are necessary to be saved.

(3) Overhead of redundant attestation. The SEV hardware has provided an attestation mechanism for remote users to verify the state of encrypted VM. However, this mechanism is a bit redundant for just verifying the integrity of restored guest system states, as it covers multiple unnecessary steps.

6.4.3 System Architecture and Workflow

Figure 6.7 depicts the architecture of my proposed system, where the `Neuralyzer` layer is integrated into confidential serverless executors. The system consists of four functional components: (1) The gateway server serves as the relay between

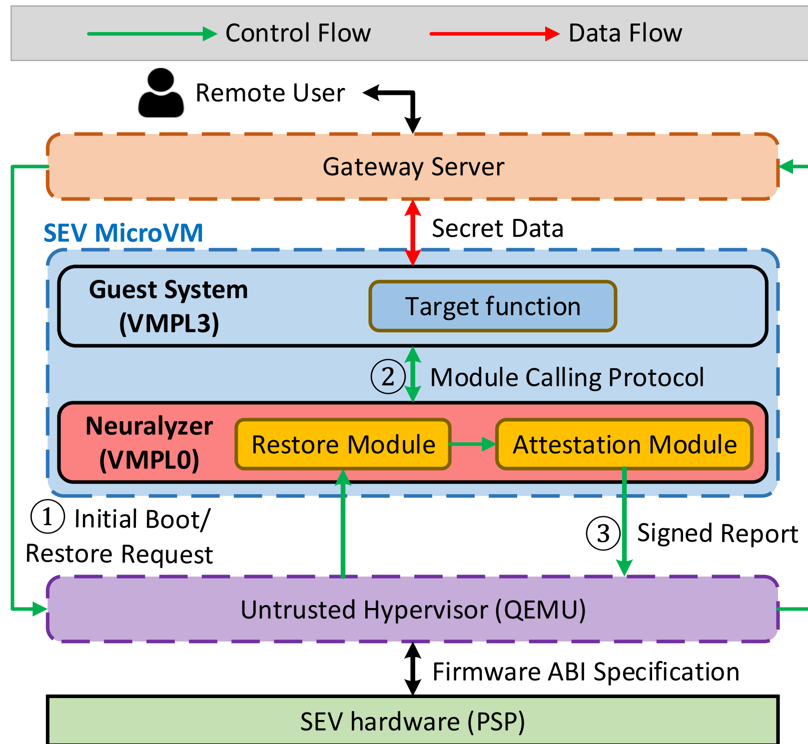


Figure 6.7: Overview of Neuralyzer architecture and workflow

remote users and backend workers; (2) The SEV VM acts as the serverless function executor; (3) The untrusted hypervisor manages guest VMs; (4) The SEV hardware serves as the Trusted Computing Base (TCB) of the confidential system. To address the above-mentioned challenges, I incorporate three key features into the `Neuralyzer` layer design: (a) I seal sensitive `Neuralyzer` modules within the privileged VMPL0 to isolate them from the untrusted guest system. (b) I perform a comprehensive analysis on the memory layout of the guest system to conduct the snapshot design for *Restore Module*. (c) I present a nested *Attestation Module* to directly verify restored states of the guest system within the VM executor. In this chapter, I adopt an SEV-compatible unikernel as the guest OS, as the adoption of unikernel is a growing trend in serverless computing [274–277]. More importantly, the simplicity of its flat and single address space can significantly reduce the difficulty of snapshot design. In theory, my design can be generalized to general-purpose operating systems, which I leave as future work.

The operation of `Neuralyzer` comprises both control flow and data flow. During the initial booting, the SEV VM is launched by the hypervisor, and an initial clean state of the guest system is saved as a snapshot buffered in the Restore Module.

Subsequent invocations of this executor prompt the hypervisor to send a restore request, which switches the VM from VMPL3 to VMPL0. The `Neuralyzer` layer shares a GHCB page with the host to receive the request (①). The Restore Module then restores the snapshot into the guest system (②). After that, the Attestation Module is called to generate a report for restored guest states and sign it with a private key (③). The signed report is sent back to the remote user through the hypervisor, and the user can verify the signature and the corresponding hash value. After the confirmation of clean guest states, the user can send the secret data through a secure channel (e.g., HTTPS).

6.5 Guest Unikernel Analysis

In this section, I first describe the design of a unikernel compatible with confidential serverless computing. Then I conduct a comprehensive analysis on its memory layout to identify necessary memory portions for the snapshot image.

6.5.1 Adaptations for Confidential Serverless

Although unikernels have been widely used in serverless computing, these implementations are often tailored and specialized for certain applications, lacking support for sophisticated hardware features, e.g., SEV. Hence, I need to design a unikernel that is compatible with SEV features. Given the complexity of SEV implementation, it demands sophisticated expertise and substantial efforts to re-implement SEV support in an entirely new unikernel framework. Consequently, I tend to strip down the latest Linux kernel supporting SEV features [300] to design a compatible unikernel. Recent works [301–303] have demonstrated the feasibility of compiling a unikernel directly from upstream Linux codebase. I make adaptations on these unikernel designs from the perspective of serverless computing and SEV support.

In the context of serverless computing, the Linux kernel, as a general monolithic kernel, includes numerous features that are redundant and unnecessary, leading to a substantial memory occupation. For example, the memory size of the unikernel compiled from UKL [301] is 148MB (with the image size of 14MB), which is too

large for serverless computing. Consequently, I customize my unikernel following four principles: (1) Reducing the kernel memory footprints and image size (e.g., disabling unnecessary features like *debug-info* and *ftrace*); (2) Retaining the features essential for the serverless scenarios; (3) Ensuring no runtime performance degradation; (4) Minimizing the possible attack surface from the untrusted hypervisor (e.g., disabling the legacy and vulnerable MMIO-based APIC and port I/O-based 8259A PIC). Finally, I effectively reduce the memory footprint of the guest unikernel by about 112 MB and the size of the kernel image by by about 7.9MB.

For the SEV support, since the latest SEV version has introduced multiple novel features (e.g., GHCB and VMPL), I need to merge them into the guest unikernel, which requires substantial modifications in the Linux codebase. In summary, the following functionalities are integrated in my unikernel: (1) *GHCB protocols support.* I add logic code for supporting GHCB protocols used to communicate with the hypervisor. (2) *VMPL detection.* I add the mechanism that enables the unikernel to read information from the *secret page* supplied by the hypervisor to detect which VMPL it is running. (3) *Module calling protocols.* I add protocol implementations in the guest unikernel to enable communication with the `Neuralyzer` layer, allowing the data transmission and delegation of privileged operations to VMPL0.

6.5.2 Memory Layout Analysis

As mentioned in the design challenge, I need to minimize the snapshot to achieve fast and lightweight serverless services, which in turn requires us to identify and only save those *necessary* memory portions. Although leveraging a unikernel has substantially reduced the complexity of analyzing the memory layout of the guest OS, the construction of a lightweight snapshot remains a challenging task. The guest unikernel preserves the standard Linux virtual address space split between user and kernel, as shown in Figure 6.8. The function stack and heap are still created in the user space, while kernel data structures (e.g., task structs, file tables) and kernel memory management services (e.g., kmalloc and vmalloc) reside in the kernel space. The function code is statically linked with the kernel code and executes in the kernel mode, meaning that the ELF segments of the function are merged into the kernel text/data segments, instead of being loaded into the user

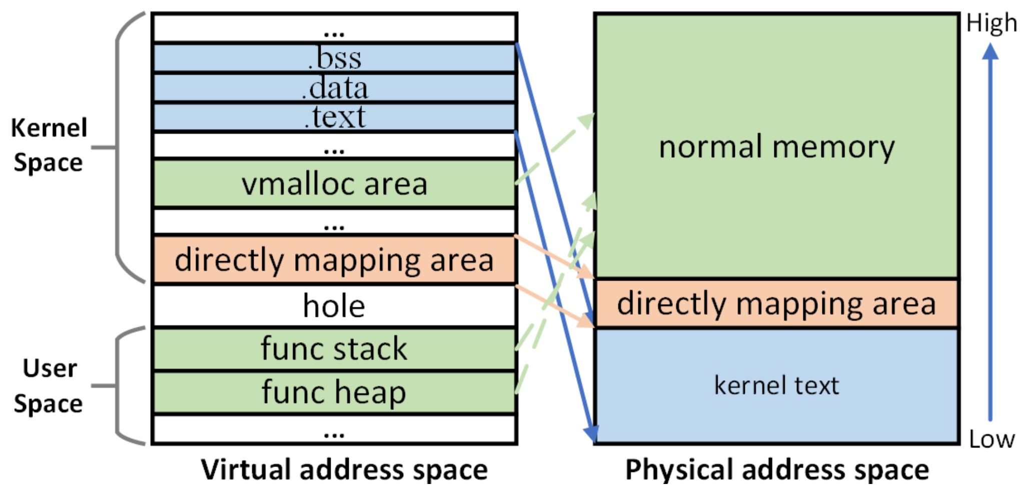


FIGURE 6.8: Memory layout of the guest unikernel

space. Consequently, the system calls between the workload function and the Linux kernel are replaced by function calls, eliminating the overhead caused by the mode switch.

The complexity of the snapshot design arises from the kernel threads spawned in the kernel space, each responsible for various tasks, e.g., scheduling or networking. As the internal attackers own the highest guest privileges, they can arbitrarily corrupt these kernel threads to craft attacks. This significantly expands the attack surface, forcing us to include both the function process and dozens of kernel threads into the protected targets that need to be restored. Fortunately, with a thorough analysis of the kernel memory layout, I find the kernel memory space is divided into various blocks and and I can simplify the snapshot design by identifying which kernel block should be saved into the snapshot:

▶ **Kernel ELF segments.** The predominant part within a Linux kernel ELF file is the `.text` segment (e.g., 28MB memory in my setting), which houses the binary code of the kernel and workload function. Adhering to OS principles, the text segment is expected to be write-protected, ensuring the integrity of this memory segment throughout the VM life cycle. So I can just save the `.data` and `.bss` segments, which contain global kernel symbols, including the kernel page tables. Note that a malicious privileged user can alter the kernel code. I address this issue by fixing the access permission on the text segment, as shown in Section 6.6.1.

▶ **Kernel data structures.** During the initiation of kernel threads, various kernel data structures would be generated and the memory allocation for these structures

is achieved by `kmalloc` with the help of slab system. These kernel structures exclusively acquire byte-sized chunks that are physically contiguous within the directly mapping area. Hence, the snapshot of data structures used in all kernel threads can be directly accomplished by saving the contiguous memory content within the directly mapping area. The start guest physical address (gPA) of this area is fixed and known by the `Neuralyzer` layer (KASLR [304] is disabled).

▶ **Kernel allocated memory.** Different from compact kernel structures, kernel stacks have much larger and fixed sizes. In recent Linux versions, the kernel stack size for x86_64 is usually 16 KB (KASAN [305] is disabled). The kernel would use `vmalloc` to allocate this large memory inside the vmalloc area. As these pages are virtually contiguous but randomly mapped to the guest physical memory, it needs to record their gPAs for subsequent snapshot saving. I achieve it by modifying the Linux kernel code. Specifically, the gPAs of allocated pages are recorded in the kernel function *__vmalloc_node_range*(), which is invoked for allocating kernel stacks and other large memory in kernel. Interrupt stacks can be directly saved as they are statically allocated.

6.6 Neuralyzer Layer Design

In this section, I present the design of the `Neuralyzer` layer for achieving secure, fast and verifiable state restoration of the SEV VM. I describe the mechanisms of the *Restore Module* and the *Attestation Module* in detail.

6.6.1 Fix Memory Access Permission

Before the design of two privileged modules, I need to address an issue proposed in Section 6.5.2, i.e., the originally write-protected `.text` segment in the guest unikernel can be altered by a malicious user sharing this cached VM. Considering the text segment typically has a large memory size of dozens of megabytes, I tend to ensure its write-protection instead of saving it into the snapshot, as the latter significantly increases the snapshot footprints. The integration of RMP and VMPL provides an opportunity to achieve this goal. Each RMP entry corresponds to a host physical page allocated to the SEV VM, recording each VMPL's access permissions

to this page, as shown in Figure 6.2. While VMPL0, i.e., the `Neuralyzer` layer, is enabled with full permissions, it can selectively grant specific permissions to lower-privileged VMPL, e.g., VMPL3, where the guest unikernel runs. The restriction of access permissions is maintained by the SEV hardware. Therefore, when VMPL0 sets the text segment in the guest unikernel as unwritable for VMPL3, even an attacker with the highest privilege within the guest OS, i.e., ring 0 in VMPL3, cannot alter the kernel code, just resulting in a #VMEXIT(NPF). To implement this, the `Neuralyzer` layer first uses the `pvalidate` instruction to validate memory pages containing the kernel text segment, whose gPA is known and fixed. Then the `rmpadjust` instruction is called to remove the write permission of VMPL3 on these pages. After that, the text segment in the guest unikernel is set as unwritable and can be protected by the SEV hardware from malicious corruptions.

6.6.2 Restore Module

The restore module is expected to remove alterations made by previous users and bring the guest OS to a known good initial state. Its overall idea follows the basic Save/Restore method: it first creates a clean snapshot by copying the guest memory and vCPU states, and later restores the snapshot by writing back the saved memory content and vCPU states. To achieve the fast and lightweight confidential serverless computing, the snapshot size should be minimized.

Snapshot Design. According to the analysis results in Section 6.5.2, the memory snapshot comprises the kernel space (i.e., the kernel ELF segments, the directly mapping area and the vmalloc area) and the user space (i.e., the function stack and heap). Among these memory portions, the `.text` segment is fixed as write-protected, so that it can be removed from the snapshot. The function stack and heap are initially empty before the execution of the workload function. Hence, I can directly set them to 0 when restoring the snapshot. Note that the canary used for protecting stack data is disabled, as it is actually ineffective in my threat model.

Consequently, the memory snapshot only needs to contain three memory portions: (1) kernel data segments (`.data` and `.bss` segments); (2) the directly mapping area containing the kernel data structures; (3) recorded kernel pages that are allocated in the vmalloc area (i.e., normal memory). Unlike the function stack, kernel stacks may contain specific data used by running kernel threads, so that they cannot be

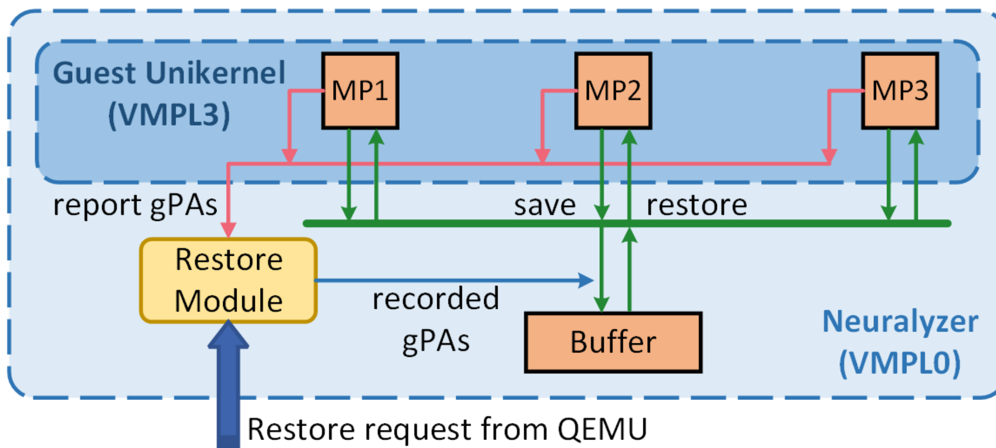


FIGURE 6.9: Workflow of restore module

directly cleaned. Considering the kernel data segments are sized in megabytes, they are much larger than the other two saved portions, which are only sized in bytes or kilobytes. Consequently, the kernel data portion constitutes the majority of the saved snapshot, which is only size of a few megabytes. Capturing the snapshot of vCPU states is a simple process. It can be achieved by directly saving the VMSA of VMPL3, which saves all register states for the guest unikernel.

Module Workflow. Figure 6.9 illustrates the workflow of the restore module. During the initial booting of the guest unikernel, the gPAs of necessary memory portions (MP) for the snapshot are recorded and subsequently reported to the `Neuralyzer` layer (red lines). Then the SEV VM switches to VMPL0 and the contents of these memory portions are read and saved to a designated buffer inside the `Neuralyzer` layer (green lines). The saving process occurs only once during the initial boot stage, which is restricted by a boolean flag. The restoring of the snapshot is triggered by a QEMU request when a new function invocation comes. The SEV VM would switch to VMPL0 to share a GHCB page with QEMU, which is used to perform protocol communication and confirm the restore request. Then the `Neuralyzer` layer restores the saved data back to the recorded gPAs, which is simply a reverse process of snapshot saving, or just sets specific memory portions (i.e., function stack and heap) to zero. The writing zero is achieved with the *rep stos* assembly code, which has been highly optimized by hardware microcode, to accelerate the process.

From the perspective of VMPL0, it owns the highest privilege within the SEV VM and consequently controls the entire memory space of the VM. Specifically,

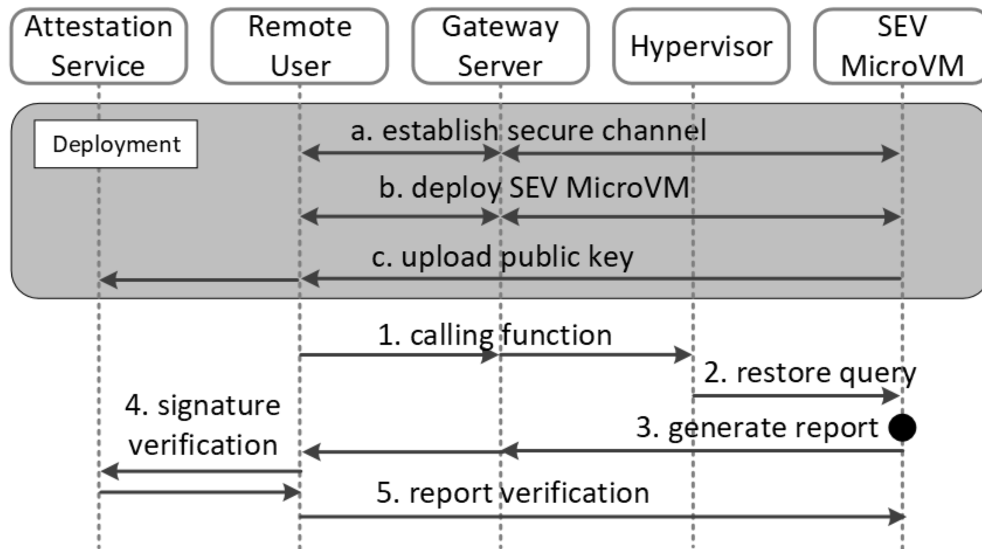


FIGURE 6.10: Workflow of attestation module

the `Neuralyzer` layer shares the same guest physical address space with the guest unikernel, while they have their own guest virtual address space. Hence, for the `Neuralyzer` layer, writing data into the guest unikernel memory is analogous to writing data into its own physical memory space, which leads to rapid operations.

6.6.3 Attestation Module

In the context of confidential serverless computing, remote users should possess the capability of verifying the integrity of restored guest OS states within the SEV VM. Although the SEV hardware has provided a remote attestation mechanism, as I mentioned before, it is too heavy for my scenario that only requires attesting the guest unikernel states. The original SEV attestation covers multiple redundant components, like the binary code of the `Neuralyzer` layer, which have already been verified during the initial booting. Since they are free from the potential attackers (i.e., attacker cannot corrupt VMPL0), I do not need to repeatedly verify them. Consequently, I tend to implement an attestation module nested in the `Neuralyzer` layer, only focusing on the guest OS states and sending the attestation report to remote users.

Module Workflow. Figure 6.10 illustrates the workflow of the attestation module. Adhering to industry practices, I employ delegated remote attestation. In this approach, a gateway server, also encapsulated within an SEV VM, conducts

attestation for all working SEV VMs and subsequently self-attests to remote users. Therefore, when remote users perform attestation to the gateway server, they can verify the confidentiality of both the gateway server and all associated SEV VMs. AT the VM deployment stage, a secure communication channel is first established to protect data transmission (step a). Then the SEV VM is deployed and the initial SEV attestation is performed (step b). Finally, a public-private key pair for signing attestation reports is generated in the attestation module. While the public key is signed by the SEV certificate chain and transmitted to an attestation service via the established secure channel, the private key remains confidential and is never shared externally (step c).

The subsequent remote user who calls this function executor will send a challenge with a *plaintext nonce* to the gateway server, which further relays the request to the hypervisor (step 1). Note that the nonce is a random number generated by the remote user, used to identify the user and mitigate possible replay attacks. The hypervisor then invokes a restore request to the `Neuralyzer` layer with the received nonce (step 2). After the restoring process is finished, the `Neuralyzer` layer will generate and sign an attestation report containing the hash of the guest OS states and the random nonce (step 3). The report is then sent to the remote user, who will verify the signature according to the public key stored in the attestation service (step 4). Finally, the remote user will validate the hash of the workload and the sent nonce to confirm if the guest VM has been restored to a known clean state (step 5). After that, the user can send the secret data through HTTPS requests or event triggers with the relaying of the gateway server.

6.7 Evaluation

6.7.1 Experimental Setup

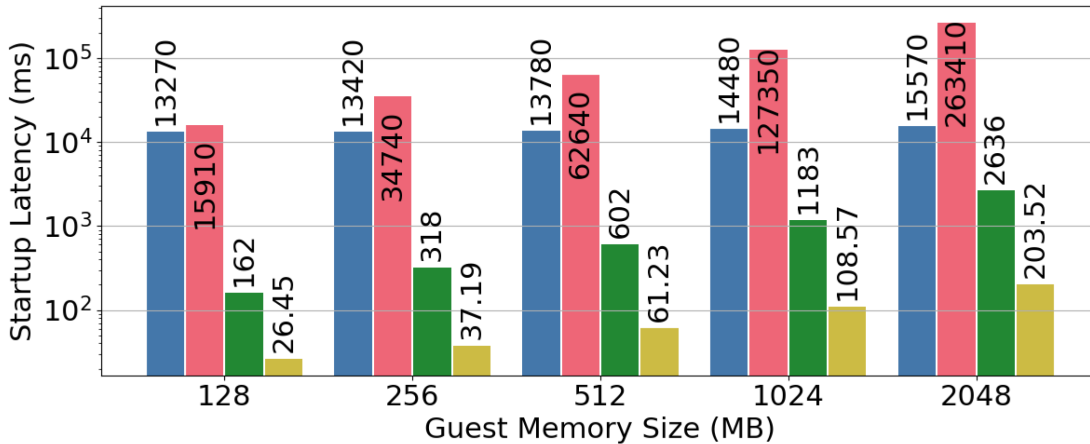
Testbed. I conduct my experiments on a server workstation with an AMD EPYC 7313P CPU (16 cores) [306] and 64 GB memory. The SEV-SNP extension and RMP memory are enabled in the host BIOS. The host OS is Ubuntu 22.04.3 with a Linux kernel version of 6.5.0 customized by AMD [300] for supporting the latest SEV features.

Implementation. The Neuralyzer system is implemented on top of five components with 4725 LoC: (1) Host Linux kernel supporting the latest SEV features (e.g., RMP and VMPL), which is directly downloaded from AMD official repository [300] and deployed on the testbed; (2) QEMU hypervisor supporting SEV API features and GHCB protocols, which is modified based on the AMD official version [307], adding 457 LoC to implement the restore query mechanism; (3) Neuralyzer layer running at VMPL0, which is modified based on the AMD proof-of-concept VMPL implementations[308, 309] with 2054 LoC, where 1483 LoC is for the restore module and 571 LoC is for the attestation module; (4) Guest unikernel modified based on the AMD's Linux branch [300] and the UKL implementation [310], where 1850 LoC is added; (5) Gateway server, containing a simple HTTPS relay service sealed in a SEV VM, which is implemented with 364 LoC. The code will be publicly available on GitHub.

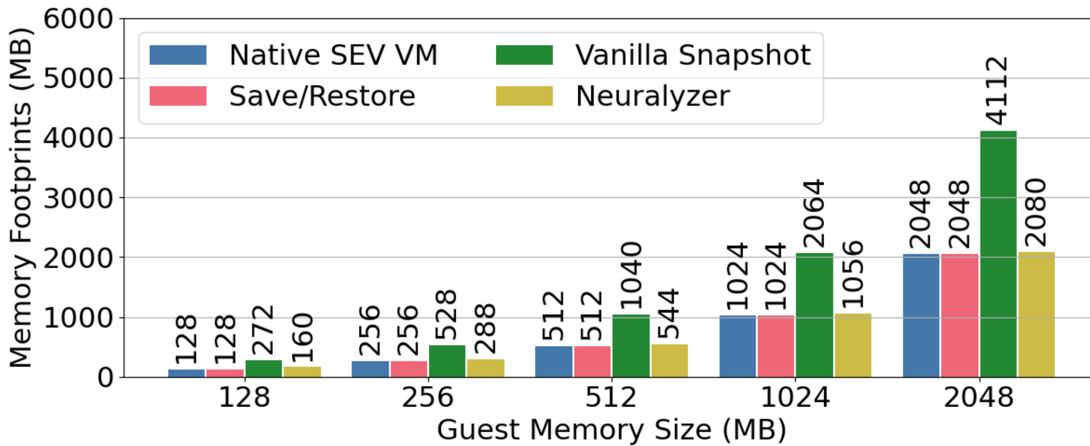
Baseline. I consider the following three systems as baselines: (1) Native SEV VM: the guest VM is directly cold-started based on the SEV extension; (2) Save/Restore: the save/restore optimization is implemented for the SEV VM using AMD official snapshot APIs (`SEND_*` and `RECEIVE_*`); (3) Vanilla snapshot: With the existence of the Neuralyzer layer, the entire guest memory is saved as the snapshot for the restore module. Note that (2) and (3) both employ the save/restore optimization, but the former performs outside the VM while the latter performs inside the VM.

6.7.2 Startup Latency

As a critical metric for assessing serverless executors, the startup latency refers to the time interval from the initialization of the executor to the moment it is ready to execute the workload function. The startup latency of a SEV VM varies depending on the guest memory size, i.e., the memory size allocated to the guest OS. Figure 6.11a illustrates the startup latency of three baselines and my Neuralyzer system. From the figure, I can see that the startup latency increases with the guest memory size. Among these four systems, the save/restore method implemented with AMD APIs performs even worse than the native SEV VM, taking minutes to restore a snapshot for the VM. This aligns with my earlier analysis in Section 6.3. As for the vanilla snapshot method, since it directly saves the entire guest memory as a snapshot, it involves the cumbersome copying of numerous unnecessary memory



(A) Startup Latency



(B) Memory Footprints

FIGURE 6.11: Comparison with three baselines

portions, such as the function stack/heap pages that can be directly set to 0. Hence, although the vanilla snapshot also can reduce the startup latency to a few hundred milliseconds, its performance decreases significantly with the increasing of guest memory size. In contrast, my `Neuralyzer` system achieves the shortest startup latency, which on average is 259× faster than the native SEV VM. The primary overhead of `Neuralyzer` arises from zeroing the function heap, while the saved kernel memory portions in the snapshot are only size of about 10MB and can be restored within 14ms.

Name	Description
helloworld	Minimal function
float_operations	Compute (sin, cos, sqrt) values
matmul	Square matrix multiplication
linpack	Solve linear equations Ax = b
image_processing	JPEG image rotation
pyaes	Text encryption with an AES block-cipher

Table 6.2: Serverless workload functions from FunctionBench

6.7.3 Memory Overhead

The fast startup of Neuralyzer comes with the price of higher memory overhead, as the SEV VM requires additional memory space for the Neuralyzer layer, which stores the module code and the saved snapshot. Figure 6.11b shows the memory footprints of the VM with varying guest memory size. The memory footprints denote the memory size occupied by the VM, including both the guest unikernel and the Neuralyzer layer (if present). From the figure, I can see that the native SEV VM and the save/restore method would not induce extra memory overhead, as their memory footprints equal to the allocated guest memory size. The vanilla snapshot method leads to about double memory footprints, as the entire guest memory is saved within the Neuralyzer layer. In contrast, Neuralyzer only has a slight memory overhead, as it just saves those necessary memory portions, which are quite small with only a few megabytes. In my implementation, the Neuralyzer layer occupies 32MB memory and is protected inside the VMPL0. Consequently, Neuralyzer achieves a good performance in terms of both startup latency and memory overhead.

6.7.4 End-to-End Performance

I present six end-to-end tests to show how Neuralyzer can reduce latency for real-world serverless functions. The workload functions are collected from the representative FunctionBench [278] suite, as shown in Table 6.2. The six workloads cover various scenarios used in serverless computing, ranging from light function calls and simple operations to heavy computation and complex image processing. The guest VM instance has a single vCPU and the guest memory size is set as 256MB, which is the typical setting in the serverless computing.

	Func Name	Time Breakdown (ms)					Total (ms)
		RR	R/L	GR	VS	EXE	
My Neuralyzer	helloworld	13.15	37.19	27.98	59.63	1.30	139.25
	float_operations	13.52	37.84	28.91	60.68	5.82	146.77
	matmul	12.91	36.93	27.63	58.32	6.17	141.96
	linpack	13.02	38.01	27.11	59.87	6.54	145.35
	image_processing	12.99	37.65	28.34	59.26	247.57	385.81
	pyaes	13.66	37.54	28.14	61.48	24.41	164.69
Native SEV VM	helloworld	0.95	13423	N/A	N/A	1.26	13425.21
	float_operations	1.01	13746	N/A	N/A	5.25	13752.26
	matmul	0.89	13584	N/A	N/A	6.93	13591.82
	linpack	1.08	13512	N/A	N/A	6.18	13519.26
	image_processing	0.93	13556	N/A	N/A	261.86	13818.79
	pyaes	0.99	13786	N/A	N/A	25.89	13812.88

TABLE 6.3: Time breakdown of end-to-end function execution, including Request Relay (RR), Restore/Launch (R/L), Generate Report (GR), Verify Signature (VS) and Execution (EXE).

To further explore the internal details of the end-to-end function execution process, I break down the execution time into five stages: Request Relay, Restore/Launch, Generate Report, Verify Signature and Execution. The time consumption for each stage is recorded and then used to compare the performance between the `Neuralyzer` system and the native SEV VMs, as shown in Table 6.3 and Figure 6.12. The results demonstrate that my system can significantly enhance the end-to-end performance, achieving an average service latency that is 84× shorter. For the native SEV VMs, the launching (R/L) stage incurs the most substantial overhead, which results in significant delays and resource wastage. In my system, the verification of signature (VS) consumes the most time, as it involves multiple steps, such as receiving the report, requesting the public key from the attestation service, and verifying the correctness of the decrypted report contents. Additionally, the request relay stage takes slightly longer time in my system. This is because it invokes the QEMU to send the restore query, resulting in a world switch from VMPL3 to VMPL0 in the guest VM, and a writing of user nonce into the GHCB page. These additional operations result in a longer latency compared to the corresponding stage in the native SEV VMs, which just requests QEMU to launch the VM.

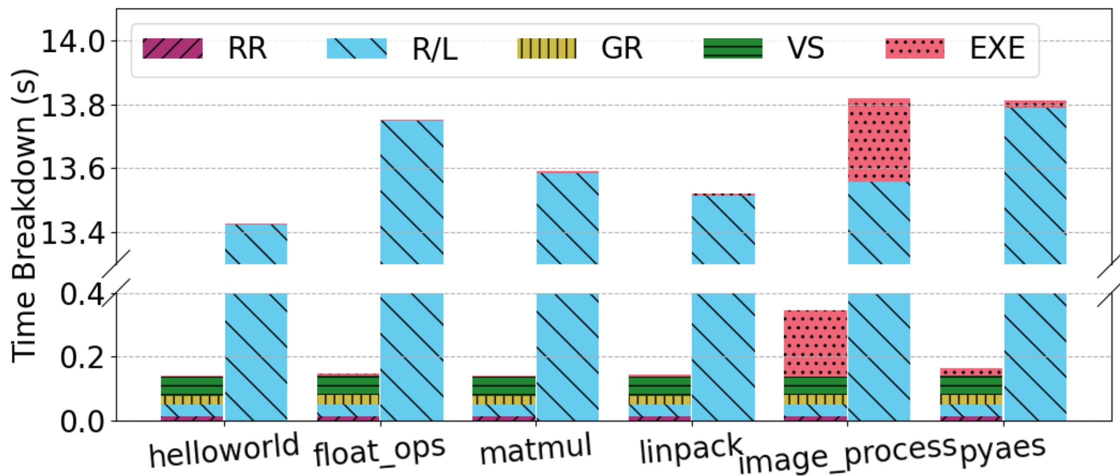


FIGURE 6.12: Time breakdown of end-to-end function execution on Neuralyzer (left bars) and native SEV VM(right bars).

6.7.5 Overhead of the Initial Booting

Unfortunately, the presence of the `Neuralyzer` layer within the guest VM increases the overhead of initial booting. Figure 6.13 shows the initial launch latency and memory footprints of a guest VM with and without the VMPL0 layer. From the figure, I can see that the `Neuralyzer` system experiences significantly longer initial booting latency compared to the native SEV VM. This is mainly caused by three additional operations introduced into the `Neuralyzer` system. The primary contributor is the startup of additional VMPL0 layer, accounting for about 90% of the increased time latency. As the controller within the VM, the `Neuralyzer` layer must be booted before the guest unikernel. Besides, the fix of access permissions on kernel text segment also brings extra time overhead, during the initial loading of the guest unikernel into the guest physical memory space. This operation involves the execution of `pvalidate` and `rmpadjust` instructions by the VMPL0 layer, which modify the host RMP table. Finally, the process of saving the snapshot from the guest unikernel into the `Neuralyzer` layer also introduces time overhead, which however is much less significant compared to the previous two operations. For the memory overhead, as mentioned in Section 6.7.3, the `Neuralyzer` layer only brings slight memory overhead, as indicated by the dashed lines in the figure.

Although the initial booting of `Neuralyzer` is heavier than the native SEV environment, it is actually a one-time overhead as all subsequent invocations execute on the same caching confidential VM. Therefore, I consider that the initial booting

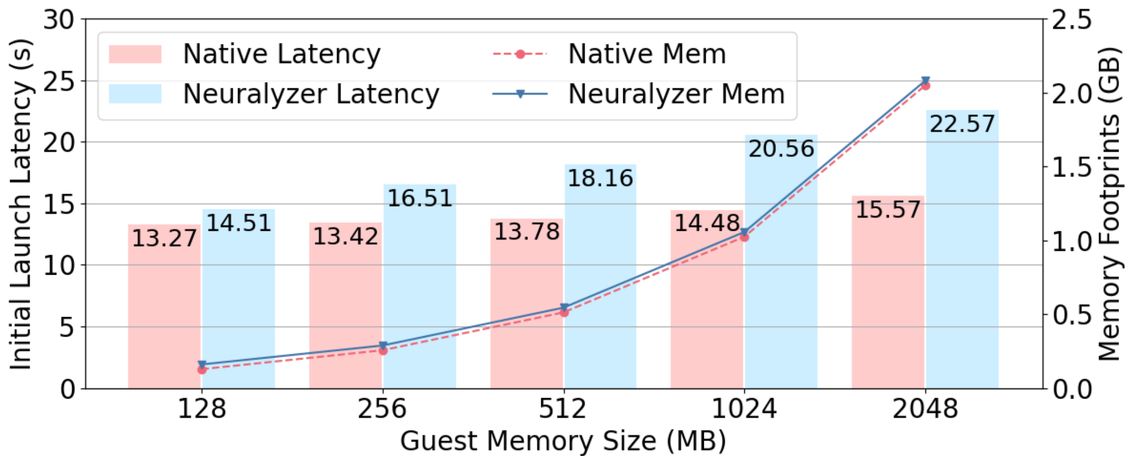


FIGURE 6.13: Initial booting performance

overhead can be easily amortized over multiple function executions, which is also a consensus in the industry [257].

6.8 Security Analysis

I perform a systematic security analysis of the `Neuralyzer` system. Specifically, I enumerate the possible attack vectors including the external attackers, internal attackers and the attacks from both sides. I describe how `Neuralyzer` is designed to defend against these attacks.

From external hypervisor. The SEV hardware has defeated most traditional attacks from the external privileged hypervisor. Here I give two representative examples:

1. Reporting false system parameters. The malicious hypervisor can report false system values (e.g., CPUID values or current time) to the guest VM, which could confuse the guest users or even corrupt the VM environment. To defend against such attack, the SEV hardware would check the inconsistency of reported system values, like validating the CPUID results stored on a special guest page or confirming current time with the SecureTSC feature [1].

2. Injecting arbitrary exceptions. The malicious hypervisor can inject arbitrary exceptions (e.g., spurious page faults) into a normal VM. However, in the SEV scenario, the hardware adds an interrupt restriction mechanism to prohibit the

hypervisor from injecting interrupts other than a newly-added #HV vector, which places all interrupts under its monitoring.

As noted in my threat model, I trust that the SEV hardware can work correctly in accordance with the official specifications to defend the confidential VM from the malicious hypervisor. Hence, I tend to pay more attention to the new attack surface introduced in my design.

3. Building snapshot during initial boot. The SEV attestation provides effective security guarantee for the initial booting of the SEV VM, as the deployment process is well measured and the initialization hash is confirmed to check if there is any mismatch with the expected results. This step ensures that the guest unikernel and `Neuralyzer` binary code are clean and perform the expected functionality, including the correct construction of the snapshot. The snapshot can only be created once during the initial booting, thereby preventing subsequent guest users from modifying its content.

4. Delaying/forging restore query. The restore module is invoked by a query from QEMU, so the malicious hypervisor can delay or even forge the query to disturb the guest VM. However, any delay or refusal to send the restore query only leads to a denial of service (DoS), as the remote user would not send the secret data before receiving the attestation report signed by the `Neuralyzer` layer. Forging a restore query would force a reset of the guest environment, where a guest user may be running. It is also a DoS attack as it blocks the function execution. The guest user only receives an error and his secret is still sealed within the VM.

5. Replay attacks. The malicious hypervisor can perform replay attacks by sending previous signed reports to cheat the remote user that the guest VM has been restored to a clean state. However, the introduced user nonce in my design can effectively defeat such attack, as it results in different hash reports for each function invocation. The replayed reports would be detected through the nonce mismatch.

From internal users. The internal attackers own the highest privileges within the guest OS, enabling them to compromise and control the entire guest OS to perform attacks:

1. Corrupting the `Neuralyzer` layer. The malicious users way want to modify the snapshot content or even the module code sealed in the `Neuralyzer` layer, so

that the restoring of clean states would fail. But the isolation between VMPLs is maintained by the SEV hardware, meaning the guest OS running at VMPL3 cannot corrupt data sealed inside VMPL0.

2. Injecting malicious code. The internal attacker may try to inject malicious code into the guest OS, aiming to embed a backdoor. Since I have fixed the write-protection of kernel text segment, the attacker cannot rewrite the kernel code.

3. Deploying code gadgets. The attacker can also craft a malicious code gadget in the guest memory and expect subsequent users to inadvertently execute it. However, my restoring mechanism overwrites all data fields, so that the attacker cannot leave any changes within the guest OS memory.

From both sides. Considering the scenario that both the guest user and hypervisor are malicious, they may attempt to cooperate to compromise the `Neuralyzer` layer. However, the VMPL0 layer is securely encrypted and isolated from both external attackers and also other internal privilege layers. Besides, the world switch between various VPMLs is well protected by the SEV hardware, making `Neuralyzer` secure even in such scenario.

6.9 Conclusion

I propose Neuralyzer, the *first* VM-based confidential serverless computing system that aims to reduce the startup latency of function executors built upon the SEV VM. By restoring clean states for cached VMs, I can significantly reduce the startup latency for a function invocation, while also ensuring the security of user secrets. My experiments show that Neuralyzer can dramatically reduce the startup latency of serverless functions and it also demonstrates satisfactory performance on real-world benchmark workloads.

Chapter 7

Conclusion and Future Work

In this chapter, I first give a summary of the work conducted in this thesis and then discuss some future research directions based on my current results.

7.1 Conclusion

Confidential computing has emerged as a critical security technology to address security and privacy challenges, making it a prominent topic in contemporary security technologies. By harnessing collaborative security in both hardware and software, it establishes a Trusted Execution Environment to guarantee confidentiality and integrity protection for data in use. As widely acknowledged, confidential computing systems are poised to serve as the next generation of computing infrastructures, applicable to emerging information technologies such as cloud computing, big data, and artificial intelligence. Unfortunately, currently proposed confidential computing systems are still in the early stages, with many vulnerable attack surfaces, particularly susceptible to threats from micro-architectural side-channel attacks. Consequently, it is urgent to perform a security analysis on existing confidential computing systems, aiming to identify those potential attack vectors and present efficient defenses. Furthermore, the design of novel confidential computing systems for recently emerging workloads is also important. In this thesis, I concentrate on addressing these issues by investigating new attack vectors, proposing unified defense frameworks and integrating sophisticated workloads to design novel systems.

This research starts with `NASPY`, a novel DNN model extraction attack targeting the sealed TEE sandbox. Model extraction attack is a conventional topic in the AI security domain, which however only focuses on hand-crafted models that just contain vanilla operations and normally requires tremendous manual analysis. My work achieves the first automatic model stealing of novel NAS models that adopt sophisticated operations without requiring any prior knowledge. Most importantly, I leverage side-channel analysis to bypass the isolation of TEE sandboxes and prove the feasibility of revealing secrets from confidential computing systems. This work shows the vulnerability of existing TEE systems with practical attacks and highlights the motivation of this thesis.

After that, I propose a unified defense framework `Aegis` to mitigate those side channels that have not been well explored but also pose significant threats, e.g., Hardware Performance Counter side channels. Through comprehensive profiling of the selected victim application, I can identify all vulnerable HPC events that could potentially leak sensitive information. Then I can conduct effective code gadgets to obfuscate these identified events with fuzzing techniques. By injecting code gadgets inside the confidential VM, the side-channel leakages can be well hidden from the malicious hypervisor, so that preventing the extraction of sensitive information.

Finally, I integrate confidential computing with emerging workloads to propose a novel confidential AI system and a novel confidential serverless computing system. The confidential AI system combines the TEE sandbox and DNN model watermark to ensure the ownership verification of IP models. I propose the first architecture-based watermark scheme that embeds watermark into the DNN model architectures. Given that such a watermark can be identified through side-channel analysis, I can verify it even when the stolen model is concealed within an encrypted black box. My watermark scheme provides great effectiveness, usability, robustness, and uniqueness for protecting DNN model ownership. Besides, I also design a novel confidential serverless computing system, which can reuse the cached TEE executor without compromising user privacy, ensuring data confidentiality while maintaining significant performance of serverless computing. Compared to simply integrating confidential computing with serverless computing, my design can significantly reduce the startup latency of the function executor by nearly hundreds of times.

7.2 Future Work

Following my dissertation research, there is still a lot more to be explored in the future.

- **More novel attack vectors.** As a sophisticated system, confidential computing systems expose a large attack surface. Apart from side-channel attacks discussed in this thesis, it still has numerous potential attack methods, which mainly can be divided into three classes: software attacks, transient execution attacks and faults injection attacks. The Iago attack [311] is a typical software attack against system calls, wherein an untrustworthy operating system attacks the confidential applications by crafting the return value of system calls. Transient execution attacks [75, 76] are actually variants of side-channel attacks, which leverage speculative execution and out-of-order execution mechanisms to obtain sensitive information. As for the faults injection attacks, e.g., Rowhammer attacks [312], they expose secret information by triggering physical or software-based faults in computations, thereby damaging the security of TEE sandboxes. All of these advanced attack vectors have the potential to be used for breaching confidential computing systems, making them worthy subjects of study in future research.
- **Defenses for other side channels.** Designing side-channel defense mechanisms for confidential computing systems is a difficult topic, as I need to simultaneously consider both performance and security. Besides, given the significant difference between various TEE techniques, i.e., process-based TEE and system-based TEE, it is impossible to propose a generic defense framework for all existing TEE sandboxes. In this thesis, my main focus is on HPC side channels, with other side channels left unexplored. While other side channels, like cache side channels, also pose significant threats to confidential computing systems, it is essential to mitigate these leakage sources to enhance the security guarantee of TEE sandboxes.
- **Application on large language models.** With the increasing popularity of Large Language Models (LLMs), integrating the LLMs serving workloads with confidential computing appears to be a promising research direction. Given that current LLMs platforms typically require users to upload their

sensitive data to online LLM APIs for inference, it exposes the privacy of user secrets to potentially malicious LLM providers. However, it is infeasible to seal the entire LLM inside a TEE sandbox, as it would significantly increase the performance overhead. Partitioning the LLM into the sensitive part and public part is a promising idea. The sensitive part is sealed in TEE sandbox for protecting security while the public part runs outside with high computing resources. I will leave it as a future work.

List of Publications

- **Xiaoxuan Lou**, Kangjie Chen, Guowen Xu, Han Qiu, Shangwei Guo, Tianwei Zhang. Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2024*.
- **Xiaoxuan Lou**, Shangwei Guo, Jiwei Li, Tianwei Zhang. Ownership Verification of DNN Architectures via Hardware Cache Side Channels. In *IEEE Transactions on Circuits and Systems for Video Technology (TCSVT), 2023.*
- **Xiaoxuan Lou**, Shangwei Guo, Jiwei Li, Yaoxin Wu, Tianwei Zhang. NASPY: Automated Extraction of Automated Machine Learning Models. in *Proceedings of the International Conference on Learning Representations (ICLR), 2022* (Spotlight Paper).
- **Xiaoxuan Lou**, Tianwei Zhang, Jun Jiang, Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. in *ACM Computing Surveys (CSUR), 2022.*
- **Xiaoxuan Lou**, Dmitrii Ustiugov, Tianwei Zhang. Enabling Fast and Secure Function Cold Starts in Confidential Serverless Systems. *Submitted to a Conference.*
- Kangjie Chen*, **Xiaoxuan Lou***, Guowen Xu, Jiwei Li, Tianwei Zhang. Clean-image Backdoor: Attacking Multi-label Models with Poisoned Labels Only. in *Proceedings of the International Conference on Learning Representations (ICLR), 2023* (Oral Paper).
- Xiaobei Yan, **Xiaoxuan Lou**, Guowen Xu, Han Qiu, Shangwei Guo, Chip Hong Chang, Tianwei Zhang. Mercury: An Automated Remote Side-channel Attack to Nvidia Deep Learning Accelerator. in *IEEE International Conference on Field-Programmable Technology (FPT), 2023.*

Bibliography

[1] A Micro Devices. Amd64 architecture programmer's manual volume 2: System programming. *2006*, 2006. xvi, 112, 134

[2] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 306–318. IEEE, 2021. xvi, 6, 17, 108, 111, 115, 116

[3] Confidential Computing Consortium et al. A technical analysis of confidential computing (v1. 1). *The Linux Foundation, San Francisco, California (confidentialcomputing. io/wp-content/uploads/sites/85/2021/03/CCC-Tech-Analysis-Confidential-Computing-V1. pdf)*, 2021. 3

[4] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016. 3, 108

[5] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016. 3, 42, 45, 112

[6] Intel® trust domain extensions (intel® tdx). [Online], . `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html`. 3, 13, 40, 42, 46, 109

[7] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, pages 217–233, 2017. 6, 15, 43

[8] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1441–1458, 2018. 6, 15, 43

[9] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017. 6, 15, 43

[10] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017. 6, 15, 43

[11] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *USENIX Security Symposium*, volume 19, pages 16–18, 2017. 6, 12, 15, 43

[12] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating branch-shadowing attacks on intel sgx using control flow randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 42–47, 2018. 6, 15, 43

[13] Jules Drean, Miguel Gomez-Garcia, Thomas Bourgeat, and Srinivas Devadas. Citadel: Side-channel-resistant enclaves with secure shared memory on a speculative out-of-order processor. *arXiv preprint arXiv:2306.14882*, 2023. 6, 16

[14] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. Darknetz: towards model privacy at the edge using trusted execution environments. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 161–174, 2020. 16

[15] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. Ppfl: privacy-preserving federated learning with trusted execution environments. In *Proceedings of the 19th annual international conference on mobile systems, applications, and services*, pages 94–108, 2021. 16

[16] Aghiles Ait Messaoud, Sonia Ben Mokhtar, Vlad Nitu, and Valerio Schiavoni. Gradsec: a tee-based scheme against federated learning inference attacks. In *Proceedings of the First Workshop on Systems Challenges in Reliable and Secure Federated Learning*, pages 10–12, 2021. 16

[17] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018. 16

[18] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. Darknight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 212–224, 2021. 16

[19] Lucien KL Ng, Sherman SM Chow, Anna PY Woo, Donald PH Wong, and Yongjun Zhao. Goten: Gpu-outsourcing trusted execution of neural network training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 14876–14883, 2021. 6, 16

[20] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 185–199, 2019. 6, 17, 108, 111

[21] Weizhong Qiang, Zezhao Dong, and Hai Jin. Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave. In *Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I*, pages 451–470. Springer, 2018. 17

[22] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*, pages 123–135, 2019. 17, 108, 111

[23] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 33–43, 2019. 17, 108

[24] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 44–54, 2019. 17

[25] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. Reusable enclaves for confidential serverless computing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4015–4032, 2023. 6, 17, 108, 111

[26] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conference*, 2006. 12

[27] Onur Aciicmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2007. 12

[28] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE Symposium on Security and Privacy*, 2015. 12

[29] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, 2007. 12

[30] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE Symposium on Security and Privacy*, 2019. 12

[31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, 2006. 12

[32] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on aes. In *International Workshop on Selected Areas in Cryptography*, 2006.

[33] Colin Percival. Cache missing for fun and profit, 2005. 12

[34] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *ACM workshop on Computer security architecture*, 2007.

[35] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *ACM conference on Computer and communications security*, 2012.

[36] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, 2015.

[37] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2016.

[38] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *IEEE Symposium on Security and Privacy*, 2015.

[39] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies*, 2017. 12

[40] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *European Workshop on Systems Security*, pages 1–6, 2017.

[41] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017. 12

[42] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M Moya. Cache misses and the recovery of the full aes 256 key. *Applied Sciences*, 2019.

[43] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2010.

[44] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Annual Design Automation Conference*, 2016.

[45] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *International Conference on Applied Cryptography and Network Security*, 2018.

[46] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2017. 12

[47] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. 2018.

[48] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017. 12, 13

[49] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ abort: A timer-free high-precision l3 cache attack using intel tsx. In *USENIX Security Symposium*, 2017. 12

[50] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *IEEE Symposium on Security and Privacy*, 2011. 12

[51] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[52] Joop van de Pol, Nigel P Smart, and Yuval Yarom. Just a little bit more. In *Cryptographers' Track at the RSA Conference*, 2015.

[53] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2014.

[54] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, 2015. 12

[55] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *Proceedings of DIMVA*, pages 279–299. Springer, 2016. 12

[56] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks. In *USENIX Security Symposium*, 2020. 12

[57] Moritz Lipp, Vedad Hažić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of amd's cache way predictors. In *ACM Asia Conference on Computer and Communications Security*, 2020. 12

[58] Wenjie Xiong and Jakub Szefer. Leaking information through cache lru states. In *IEEE International Symposium on High Performance Computer Architecture*, 2020. 12

[59] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 2017. 12

[60] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 2019. 12

[61] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. In *USENIX Security Symposium*, 2018. 12

[62] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *ACM SIGSAC Conference on Computer and Communications Security*, 2019. 12

[63] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015. 12, 13

[64] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *ACM on Asia Conference on Computer and Communications Security*, 2016.

[65] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[66] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page {Table-Based} attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, 2017. 12, 13

[67] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*, 2016. 12

[68] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy*, 2020. 12

[69] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, 1996. 11

[70] Daniel J Bernstein. Cache-timing attacks on aes. *Technical Report*, 2005. 11

[71] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2006. 11

[72] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *International cryptology conference*, 2014. 11

[73] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs. *Journal of Cryptographic Engineering*, 2015. 11

[74] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 1999. 11

[75] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018. 12, 73, 139

[76] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020. 12, 118, 139

[77] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015. 12, 13, 42

[78] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014. 12, 94

[79] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *arXiv preprint arXiv:1810.03487*, 2018. 13, 80, 88

[80] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security Symposium*, pages 2003–2020, 2020. 13, 22, 23, 25, 26, 28, 32, 40, 50, 79, 80, 88, 90, 91, 93

[81] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security Symposium*, pages 515–532, 2019. 13, 80

[82] Vasisht Duddu, Debasis Samanta, D Vijay Rao, and Valentina E Balas. Stealing neural networks via timing side channels. *arXiv preprint*, 2018. 13

[83] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. DeepSniffer: A DNN model extraction framework based on learning architectural hints. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 385–399, 2020. 13, 16, 22, 23, 25, 27, 28, 31, 32, 80, 100

[84] Weizhe Hua, Zhiru Zhang, and G Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018. 13, 22

[85] Amdsev snp-latest branch. `https://github.com/AMDESE/AMDSEV/tree/snp-latest`. 13, 114

[86] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017. 13

[87] Jan Werner, Joshua Mason, and et al. The severest of them all: Inference attacks against secure virtual enclaves. In *Proceedings of ACM AsiaCCS*, 2019. 13, 42, 47

[88] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected I/O operations in AMD's secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1257–1272, 2019. 13, 42

[89] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. Severity: Code injection attacks against encrypted virtual machines. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 444–455. IEEE, 2021. 13, 42

[90] Mathias Morbitzer, Manuel Huber, and et al. Severed: Subverting amd's virtual machine encryption. In *Proceedings of EuroSec*, 2018. 13, 42

[91] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In *ACM SIGPLAN Notices*, 2017.

[92] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking" security-by-crash" based memory isolation in amd sev. In *Proceedings of ACM CCS*, 2021. 13, 42, 47, 118

[93] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *IEEE/IFIP Intl. Conf. on Dependable Systems and Networks Workshops*, 2011. 14

[94] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Conf. on Security Symposium*, 2012. 14

[95] Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *ACM Conf. on Computer and Communications Security*, 2013. 14

[96] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *IEEE Sixth International Conference on Cloud Computing*, 2013. 14

[97] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-vm side-channels. In *USENIX Security Symposium*, 2014. 14

[98] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *IEEE International Conference on Cloud Engineering*, 2018. 14

[99] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1992. 14

[100] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *ACM Workshop on Cloud Computing Security*, 2011. 14

[101] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Annual International Symposium on Computer Architecture*, 2012. 15

[102] Peng Li, Debin Gao, and Michael K. Reiter. Stopwatch: A cloud architecture for timing channel mitigation. *ACM Trans. Inf. Syst. Secur.*, 2014. 15

[103] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Network and Distributed System Security Symposium*, 2015. 15

[104] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015. 15

[105] Hai Huang, Jiaming Mu, Neil Zhenqiang Gong, Qi Li, Bin Liu, and Mingwei Xu. Data poisoning attacks to deep learning based recommender systems. *arXiv preprint arXiv:2101.02644*, 2021. 16

[106] Matthew Jagielski, Giorgio Severi, Niklas Pousette Harger, and Alina Oprea. Subpopulation data poisoning attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3104–3122, 2021.

[107] Kangjie Chen, Yuxian Meng, Xiaofei Sun, Shangwei Guo, Tianwei Zhang, Jiwei Li, and Chun Fan. Badpre: Task-agnostic backdoor attacks to pre-trained nlp foundation models. *arXiv preprint arXiv:2110.02467*, 2021.

[108] Kangjie Chen, Xiaoxuan Lou, Guowen Xu, Jiwei Li, and Tianwei Zhang. Clean-image backdoor: Attacking multi-label models with poisoned labels only. In *The Eleventh International Conference on Learning Representations*, 2022. 16

[109] Milad Nasr, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A Feder Cooper, Daphne Ippolito, Christopher A Choquette-Choo, Eric Wallace, Florian Tramèr, and Katherine Lee. Scalable extraction of training data from (production) language models. *arXiv preprint arXiv:2311.17035*, 2023. 16

[110] Chuan Guo, Brian Karrer, Kamalika Chaudhuri, and Laurens van der Maaten. Bounding training data reconstruction in private (deep) learning. In *International Conference on Machine Learning*, pages 8056–8071. PMLR, 2022.

[111] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. {Updates-Leak}: Data set inference and reconstruction attacks in online learning. In *29th USENIX security symposium (USENIX Security 20)*, pages 1291–1308, 2020.

[112] Haomiao Yang, Mengyu Ge, Kunlan Xiang, and Jingwei Li. Using highly compressed gradients in federated learning for data reconstruction attacks. *IEEE Transactions on Information Forensics and Security*, 18:818–830, 2022. 16

[113] Wenbo Jiang, Hongwei Li, Guowen Xu, Tianwei Zhang, and Rongxing Lu. A comprehensive defense framework against model extraction attacks. *IEEE Transactions on Dependable and Secure Computing*, 2023. 16

[114] Xueluan Gong, Yanjiao Chen, Wenbin Yang, Guanghao Mei, and Qian Wang. Inversenet: Augmenting model extraction attacks with training data inversion. In *IJCAI*, pages 2439–2447, 2021. 16

[115] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: functional encryption using intel sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 765–782, 2017. 17

[116] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018. 17

[117] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. Azure sql database always encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1511–1525, 2020. 17

[118] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, et al. Operon: An encrypted database for ownership-preserving data management. *Proceedings of the VLDB Endowment*, 15(12):3332–3345, 2022. 17

[119] Jinwei Zhu, Kun Cheng, Jiayang Liu, and Liang Guo. Full encryption: An end to end encryption mechanism in gaussdb. *Proceedings of the VLDB Endowment*, 14(12):2811–2814, 2021. 17

[120] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019. 17

[121] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 22, 24, 79

[122] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019. 22, 24

[123] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Dana Dachman-Soled, and Tudor Dumitraş. How to 0wn NAS in your spare time. *arXiv preprint arXiv:2002.06776*, 2020. 22, 23, 25, 26, 28, 40, 79

[124] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 321–338, 2019. 22

[125] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In *International Conference on Machine Learning*, pages 2137–2146. PMLR, 2018. 22

[126] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017. 22

[127] Seong Joon Oh, Bernt Schiele, and Mario Fritz. Towards reverse-engineering black-box neural networks. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pages 121–144. Springer, 2019. 22

[128] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2139–2153, 2018. 22

[129] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018. 23, 24, 34, 39, 79, 86, 89, 90, 91, 92

[130] Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four GPU hours. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1761–1770, 2019. 23, 24, 34, 79, 90, 91, 92, 94

[131] Wuyang Chen, Xinyu Gong, and Zhangyang Wang. Neural architecture search on imagenet in four gpu hours: A theoretically inspired perspective. *arXiv preprint arXiv:2102.11535*, 2021. 23, 34

[132] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018. 24, 79, 86, 89, 90, 92, 94

[133] Xuanyi Dong and Yi Yang. NAS-Bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020. 24, 34, 89

[134] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018. 24, 79

[135] Xiangxiang Chu, Bo Zhang, Ruijun Xu, and Jixiang Li. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *arXiv preprint arXiv:1907.01845*, 2019. 24

[136] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 550–559, 2018. 24

[137] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *AAAI Conference on Artificial Intelligence*, volume 33, pages 4780–4789, 2019. 24, 90, 92

[138] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv preprint arXiv:1804.09081*, 2018. 24

[139] Xiangxiang Chu, Tianbao Zhou, Bo Zhang, and Jixiang Li. Fair DARTS: Eliminating unfair advantages in differentiable architecture search. In *European Conference on Computer Vision*, pages 465–480, 2020. 24, 90

[140] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. Hmtt: A hybrid hardware/software tracing system for bridging the dram access trace's semantic gap. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(1):1–25, 2014. 25, 39

[141] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182. PMLR, 2016. 26

[142] Graham Neubig. Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*, 2017. 26

[143] Md Sanzidul Islam, Sadia Sultana Sharmin Mousumi, Sheikh Abujar, and Syed Akhter Hossain. Sequence-to-sequence bangla sentence generation with lstm recurrent neural networks. *Procedia Computer Science*, 152:51–58, 2019. 26

[144] Kulothunkan Palasundram, Nurfadhlina Mohd Sharef, Khairul Azhar Kasmiran, and Azreen Azman. Enhancements to the sequence-to-sequence-based natural answer generation models. *IEEE Access*, 8:45738–45752, 2020. 26

[145] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High accuracy and high fidelity extraction of neural networks. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1345–1362, 2020. 26

[146] Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3): 1–25, 2008. 29

[147] Daniel S Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D Cubuk, and Quoc V Le. Specaugment: A simple data augmentation method for automatic speech recognition. *arXiv preprint arXiv:1904.08779*, 2019. 31

[148] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. 32

[149] Ming Lin, Pichao Wang, Zhenhong Sun, Hesen Chen, Xiuyu Sun, Qi Qian, Hao Li, and Rong Jin. Zen-nas: A zero-shot nas for high-performance deep image recognition. *arXiv preprint arXiv:2102.01063*, 2021. 39

[150] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310, 2013. 40

[151] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 675–692, 2019. 40

[152] Tan Qinhan, Zeng Zhihua, Bu Kai, et al. Phantomcache: Obfuscating cache conflicts with localized randomization. In *Proc of the 2020 NDSS Symp. San Diego, CA: ISOC*, 2020. 40

[153] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*, volume 1, pages 57–64. IEEE, 2015. 42

[154] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted execution environments: properties, applications, and challenges. *IEEE Security & Privacy*, 18(2):56–60, 2020. 42

[155] Amd expands confidential computing presence on google cloud. [Online], . `https://www.amd.com/en/press-releases/2022-05-25-amd-expands-confidential-computing-presence-google-cloud`. 42

[156] Confidential computing: an aws perspective. [Online], . `https://aws.amazon.com/blogs/security/confidential-computing-an-aws-perspective/`. 42

[157] Arm confidential compute architecture. [Online]. `https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture`. 42, 46, 109

[158] Amd64 architecture programmer's manual, volume 2: System programming. [Online], . `https://www.amd.com/system/files/TechDocs/24593.pdf`. 42

[159] AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020. 42, 43, 46, 112, 114

[160] Sanjeev Das, Bihuan Chen, and et al. Ropsentry: Runtime defense against rop attacks using hardware performance counters. *Computers & Security*, 73:374–388, 2018. 43

[161] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *Proceedings of RAID*, pages 118–140. Springer, 2016.

[162] Xueyang Wang and Ramesh Karri. Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(3): 485–498, 2015. 43

[163] Ning Zhang, Kun Sun, and et al. Truspy: Cache side-channel information leakage from the secure world on arm devices. *IACR Cryptol. ePrint Arch.*, 2016:980, 2016. 43, 45

[164] Moritz Lipp, Daniel Gruss, and et al. Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, 2016. 45

[165] Berk Gulmezoglu, Andreas Zankl, and et al. Perfweb: How to violate web privacy with hardware performance events. In *Proceedings of ESORICS*, pages 80–97, 2017. 45, 47

[166] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *Proceedings of FDTC-Workshop*, 2008. 45

[167] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen?: Utilizing performance monitors for compromising keys of rsa on intel platforms. In *Proceedings of CHES*, pages 248–266. Springer, 2015. 43, 45, 47

[168] Intel tdx module specification 1.5. [Online], . `https://cdrdv2.intel.com/v1/dl/getContent/733575`. 43

[169] Azure confidential vm options. [Online], . `https://learn.microsoft.com/en-us/azure/confidential-computing/virtual-machine-solutions`. 43

[170] Google cloud confidential vm overview. [Online], . `https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview`. 43

[171] Amazon ec2 user guide. [Online], . `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html`. 43

[172] Azure linux virtual machines pricing. [Online], . `https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/linux/`. 43

[173] Google confidential vm supported configurations. [Online], . `https://cloud.google.com/confidential-computing/confidential-vm/docs/supported-configurations`. 43

[174] David Kaplan. Protecting vm register state with sev-es. *White paper*, 2017. 46

[175] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019. 46

[176] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019. 46

[177] Sam Hocevar. Zzuf. [Online]. `https://github.com/samhocevar/zzuf/`. 46

[178] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, 2019. 46

[179] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019. 46

[180] Christopher Domas. Breaking the x86 isa. *Black Hat*, 2017. 46

[181] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1415–1432, 2021. 46

[182] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. Speechminer: A framework for investigating and measuring speculative execution vulnerabilities. *arXiv preprint arXiv:1912.00329*, 2019. 46

[183] M Caner Tol, Berk Gulmezoglu, Koray Yurtseven, and Berk Sunar. Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 616–632. IEEE, 2021. 46

[184] Cynthia Dwork. Differential privacy. In *Proceedings of ICALP*, pages 1–12. Springer, 2006. 46

[185] Konstantinos Chatzikokolakis, Miguel E Andrés, and et al. Broadening the scope of differential privacy using metrics. In *Proceedings of PETs*, 2013. 46

[186] Qiuyu Xiao, Michael K Reiter, and Yinqian Zhang. Mitigating storage side channels using statistical privacy mechanisms. In *Proceedings of ACM CCS*, pages 1582–1594, 2015. 47, 50, 62, 64

[187] Xiaokuan Zhang, Jihun Hamm, and et al. Statistical privacy for streaming traffic. In *Proceedings of NDSS*, 2019. 47, 62

[188] Debayan Das, Anupam Golder, Josef Danial, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. X-deepsca: Cross-device deep learning side channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019. 48

[189] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Analyzing cache side channels using deep neural networks. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 174–186, 2018. 48

[190] Amd secure encrypted virtualization (sev) github repository. [Online], . `https://github.com/AMDESE/AMDSEV`. 48

[191] Payap Sirinam, Mohsen Imani, and et al. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proceedings of ACM CCS*, 2018. 48

[192] Wladimir De la Cadena, Asya Mitseva, and et al. Trafficsliver: Fighting website fingerprinting attacks with traffic splitting. In *Proceedings of ACM CCS*, 2020.

[193] Anatoly Shusterman, Lachlan Kang, and et al. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security Symposium*, 2019. 48

[194] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015. 48

[195] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014. 49

[196] Alexa top 1000 most visited websites. [Online]. `https://www.htmlstrip.com/alexa-top-1000-most-visited-websites`. 49

[197] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001. 49, 55

[198] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *2012 IEEE Symposium on Security and Privacy*, pages 143–157. IEEE, 2012. 49, 50

[199] Ubuntu manpage for xdotool. [Online]. `https://manpages.ubuntu.com/manpages/trusty/man1/xdotool.1.html`. 50

[200] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, pages 601–618, 2016. 50

[201] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. Open dnn box by power side-channel attack. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(11):2717–2721, 2020. 50

[202] Xiaoxuan Lou, Shangwei Guo, Jiwei Li, Yaoxin Wu, and Tianwei Zhang. Naspy: Automated extraction of automated machine learning models. In *International Conference on Learning Representations*, 2021. 50

[203] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006. 50

[204] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014. 50

[205] Pytorch models and pretrained weights. [Online]. `https://pytorch.org/vision/stable/models.html`. 50

[206] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2019. 52

[207] perfmon2 libpfm-4.11.0 released. [Online], . `http://perfmon2.sourceforge.net/`. 54

[208] Linux kernel profiling with perf: multiplexing and scaling events. [Online]. `https://perf.wiki.kernel.org/index.php/Tutorial#multiplexing_and_scaling_events`. 54

[209] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901. 55

[210] Ramanathan Gnanadesikan and Martin B Wilk. Probability plotting methods for the analysis of data. *Biometrika*, 55(1):1–17, 1968. 55

[211] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 673–686, 2019. 58

[212] T-H Hubert Chan, Elaine Shi, and et al. Private and continual release of statistics. *ACM Transactions on Information and System Security*, 2011. 63

[213] Eloi de Chérisey, Sylvain Guilley, Olivier Rioul, and Pablo Piantanida. Best information is most successful. *Cryptology ePrint Archive*, 2019. 70, 73

[214] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019. 73

[215] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2875–2889, 2021. 73, 117, 118

[216] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25:1097–1105, 2012. 78

[217] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE CVPR*, 2016.

[218] Shangwei Guo, Tianwei Zhang, Guowen Xu, Han Yu, Tao Xiang, and Yang Liu. Topology-aware differential privacy for decentralized image classification. *IEEE Transactions on Circuits and Systems for Video Technology*, 2021. 78

[219] Yang Wang, Xiaopeng Fan, Ruiqin Xiong, Debin Zhao, and Wen Gao. Neural network-based enhancement to inter prediction for video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(2):826–838, 2021. 78

[220] Linghui Li, Yongdong Zhang, Sheng Tang, Lingxi Xie, Xiaoyong Li, and Qi Tian. Adaptive spatial location with balanced loss for video captioning. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(1): 17–30, 2022. 78

[221] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 78

[222] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. 78

[223] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020. 78

[224] Lixin Luo, Zhenyong Chen, Ming Chen, Xiao Zeng, and Zhang Xiong. Reversible image watermarking using interpolation technique. *IEEE Transactions on Information Forensics and Security*, 5(1):187–193, 2009. 78

[225] Aniket Roy and Rajat Subhra Chakraborty. Toward optimal prediction error expansion-based reversible image watermarking. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(8):2377–2390, 2019.

[226] Han Fang, Dongdong Chen, Qidong Huang, Jie Zhang, Zehua Ma, Weiming Zhang, and Nenghai Yu. Deep template-based watermarking. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(4):1436–1451, 2020.

[227] Qi Li, Xingyuan Wang, Bin Ma, Xiaoyu Wang, Chunpeng Wang, Suo Gao, and Yunqing Shi. Concealed attack for robust watermarking based on generative model and perceptual loss. *IEEE Transactions on Circuits and Systems for Video Technology*, 2021.

[228] Lizhi Xiong, Xiao Han, Ching-Nung Yang, and Yun-Qing Shi. Robust reversible watermarking in encrypted image with secure multi-party based on lightweight cryptography. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(1):75–91, 2021.

[229] Jinkun You, Yuan-Gen Wang, Guopu Zhu, and Sam Kwong. Truncated robust natural watermarking with hungarian optimization. *IEEE Transactions on Circuits and Systems for Video Technology*, 2021.

[230] Fei Peng, Bo Long, and Min Long. A general region nesting-based semi-fragile reversible watermarking for authenticating 3d mesh models. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(11):4538–4553, 2021. 78

[231] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. In *ACM on International Conference on Multimedia Retrieval*, pages 269–277, 2017. 78, 81

[232] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. DeepSigns: An end-to-end watermarking framework for protecting the ownership of deep neural networks. In *ACM ASPLOS*, 2019. 81

[233] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *USENIX Security Symposium*, pages 1615–1631, 2018. 78, 81, 83, 100

[234] Jie Zhang, Dongdong Chen, Jing Liao, Han Fang, Weiming Zhang, Wenbo Zhou, Hao Cui, and Nenghai Yu. Model watermarking for image processing networks. In *AAAI Conference on Artificial Intelligence*, volume 34, pages 12805–12812, 2020.

[235] Kangjie Chen, Shangwei Guo, Tianwei Zhang, Shuxin Li, and Yang Liu. Temporal watermarks for deep reinforcement learning models. In *International Conference on Autonomous Agents and Multiagent Systems*, 2021. 100

[236] Hanzhou Wu, Gen Liu, Yuwei Yao, and Xinpeng Zhang. Watermarking neural networks with watermarked images. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(7):2591–2601, 2020. 78

[237] Xinyun Chen, Wenxiao Wang, Chris Bender, Yiming Ding, Ruoxi Jia, Bo Li, and Dawn Song. REFIT: A unified watermark removal framework for deep learning systems with limited data. *ACM AsiaCCS*, 2021. 78, 100

[238] Masoumeh Shafieinejad, Jiaqi Wang, Nils Lukas, Xinda Li, and Florian Kerschbaum. On the robustness of the backdoor-based watermarking in deep neural networks. *arXiv preprint arXiv:1906.07745*, 2019.

[239] Xuankai Liu, Fengting Li, Bihan Wen, and Qi Li. Removing backdoor-based watermarks in neural networks with limited data. *arXiv preprint arXiv:2008.00407*, 2020. 78

[240] Shangwei Guo, Tianwei Zhang, Han Qiu, Yi Zeng, Tao Xiang, and Yang Liu. Fine-tuning is not enough: A simple yet effective watermark removal attack for DNN models. *International Joint Conference on Artificial Intelligence*, 2021. 78, 100

[241] Ryota Namba and Jun Sakuma. Robust watermarking of neural network with exponential weighting. In *ACM AsiaCCS*, 2019. 79, 81

[242] William Aiken, Hyoungshick Kim, and Simon Woo. Neural network laundering: Removing black-box backdoor watermarks from deep neural networks. *arXiv preprint arXiv:2004.11368*, 2020. 79

[243] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017. 79, 89

[244] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. SMASH: One-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017. 79

[245] Erwan Le Merrer, Patrick Perez, and Gilles Trédan. Adversarial frontier stitching for remote neural network watermarking. *Neural Computing and Applications*, pages 1–12, 2019. 81

[246] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. Protecting intellectual property of deep neural networks with watermarking. In *ACM AsiaCCS*, 2018. 81, 83

[247] Zheng Li, Chengyu Hu, Yang Zhang, and Shanqing Guo. How to prove your model belongs to you: A blind-watermark based framework to protect intellectual property of DNN. In *Annual Computer Security Applications Conference*, pages 126–137, 2019. 81

[248] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *European Conference on Computer Vision*, pages 19–34, 2018. 92

[249] Lei Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? *arXiv preprint arXiv:1312.6184*, 2013. 102

[250] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 102

[251] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016. 103

[252] Aws lambda. `https://aws.amazon.com/lambda/`, . 108

[253] Azure functions. `https://azure.microsoft.com/en-us/products/functions/`, . 108

[254] Google cloud functions. `https://cloud.google.com/functions/`, . 108

[255] Eduard Marin, Diego Perino, and Roberto Di Pietro. Serverless computing: a security perspective. *Journal of Cloud Computing*, 11(1):1–12, 2022. 108

[256] Carlos Segarra, Tobin Feldman-Fitzthum, Daniele Buono, and Peter Pietzuch. Serverless confidential containers: Challenges and opportunities. In *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies*, pages 32–40, 2024. 108

[257] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020. 108, 109, 111, 115, 134

[258] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018. 108

[259] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR)*, 54(10s):1–34, 2022. 108

[260] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020. 108, 113, 115

[261] Google gvisor. `https://gvisor.dev/`. 108, 113

[262] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. {RunD}: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, 2022. 108

[263] Cloud hypervisor. `https://www.cloudhypervisor.org/`. 108

[264] Intel 12th generation intel® core™ processors datasheet. `https://cdrdv2.intel.com/v1/dl/getContent/655258`, . 108

[265] Amd secure encrypted virtualization (sev). `https://www.amd.com/en/developer/sev.html`, . 109, 112

[266] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019. 109, 117

[267] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021.

[268] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020. 109, 116

[269] Lixiang Ao, George Porter, and Geoffrey M Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 730–746, 2022. 109, 117

[270] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards {High-Performance} serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018. 109, 116

[271] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM international conference on architectural support for programming languages and operating systems*, pages 797–813, 2022. 109, 116

[272] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021. 109, 115

[273] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 57–70, 2018. 109, 115

[274] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020. 110, 113, 120

[275] Bo Tan, Haikun Liu, Jia Rao, Xiaofei Liao, Hai Jin, and Yu Zhang. Towards lightweight serverless computing via unikernel as a function. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2020.

[276] Henrique Fingler, Amogh Akshintala, and Christopher J Rossbach. Usetl: Unikernels for serverless extract transform and load why should you settle for less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 23–30, 2019.

[277] Chetankumar Mistry, Bogdan Stelea, Vijay Kumar, and Thomas Pasquier. Demonstrating the practicality of unikernels to build a serverless platform at the edge. In *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 25–32. IEEE, 2020. 110, 113, 120

[278] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019. 110, 131

[279] Confidential Computing Consortium. Confidential computing: Hardware-based trusted execution for applications and data. `https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC_outreach_whitepaper_updated_November_2022.pdf`. 111

[280] Kata containers with amd sev-snp vms. `https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/how-to-run-kata-containers-with-SNP-VMs.md`. 111, 114

[281] Attestable, confidential workloads with libkrun and amd sev-snp. `https://virtee.io/attestable-confidential-workloads-libkrun/`, . 111, 114

[282] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 30–44, 2020. 111

[283] David Kaplan. Protecting vm register state with sev-es. *White paper*, page 13, 2017. 112

[284] Sev-es guest hypervisor communication block (ghcb) standardization. `https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56421.pdf`. 112

[285] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? *Queue*, 11 (11):30–44, 2013. 113

[286] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017. 113

[287] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 169–173, 2017. 113

[288] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. Fades: Fine-grained edge offloading with unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, pages 36–41, 2017. 113

[289] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu:{Just-In-Time} summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, 2015. 113

[290] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015. 113

[291] Chia-Che Tsai, Donald E Porter, and Mona Vij. {Graphene-SGX}: A practical library {OS} for unmodified applications on {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017. 113

[292] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast {RDMA-codesigned} remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, 2023. 116

[293] Third sgx community day. `https://community.intel.com/t5/Blogs/Tech-Innovation/Data-Center/Third-SGX-Community-Day/post/1393177`. 116

[294] Secure encrypted virtualization api. `https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/55766_SEV-KM_API_Specification.pdf`, . 117

[295] Jianqiang Wang, Pouya Mahmoody, Ferdinand Brasser, Patrick Jauernig, Ahmad-Reza Sadeghi, Donghui Yu, Dahan Pan, and Yuanyuan Zhang. Virtee: A full backward-compatible tee with native live migration and secure i/o. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 241–246, 2022. 117

[296] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004. 118

[297] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Pwrleak: Exploiting power reporting interface for side-channel attacks on amd sev. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–66. Springer, 2023. 118

[298] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking constant-time cryptography on AMDSEV via the ciphertext side channel. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 717–732, 2021. 118

[299] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. Sevurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1483–1496. IEEE, 2020. 118

[300] The latest sev-snp linux branch. `https://github.com/AMDESE/linux/tree/snp-host-latest`, . 121, 128, 129

[301] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, et al. Unikernel linux (ukl). In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 590–605, 2023. 121

[302] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. Unikernels: The next stage of linux's dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 7–13, 2019.

[303] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020. 121

[304] Kernel address space layout randomization (kaslr). `https://lwn.net/Articles/569635/`, . 124

[305] Kernel address sanitizer (kasan). `https://docs.kernel.org/dev-tools/kasan.html`, . 124

[306] Amd epyc™ 7313p cpu. `https://www.amd.com/en/products/cpu/amd-epyc-7313p`, . 128

[307] Qemu branch for sev-snp. `https://github.com/AMDESE/qemu/tree/snp-latest`, . 129

[308] Linux svsm (secure vm service module). `https://github.com/AMDESE/linux-svsm/`, . 129

[309] Coconut secure vm service module. `https://github.com/coconut-svsm/svsm`. 129

[310] Unikernel linux (ukl). `https://github.com/unikernelLinux/ukl`. 129

[311] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013. 139

[312] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014. 139