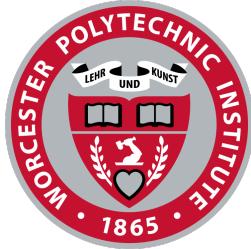


AI-Assisted Detection and Mitigation of Microarchitectural Vulnerabilities

Muttalip Caner Tol



A Dissertation
Submitted to the Faculty
of the

WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy
in
Electrical and Computer Engineering

December 2024

APPROVED:

Professor Stjepan Picek, Committee Member, Radboud University

Professor Ziming Zhang, Committee Member, ECE, Worcester Polytechnic Institute

Professor Berk Sunar, Advisor, ECE, Worcester Polytechnic Institute

Professor Donald Brown, Department Head, ECE, Worcester Polytechnic Institute

Abstract

Microarchitectural vulnerabilities pose a significant security challenge, enabling attackers to exploit hardware optimizations to compromise system integrity and data confidentiality. These vulnerabilities, often hidden in the complex interactions between modern processors, memory hierarchies and software stack. It is required to have innovative techniques for their discovery, and mitigation. Existing approaches struggle to address the scale and complexity of these threats, necessitating new methodologies that combine the strengths of artificial intelligence and optimization.

This dissertation presents a comprehensive exploration of microarchitectural vulnerabilities, leveraging novel AI-driven techniques to tackle some of the most pressing challenges in the field. On the fault attack side, we demonstrate a Rowhammer-based backdoor injection attack on machine learning models deployed on real hardware. This work introduces a constrained optimization framework to efficiently identify and exploit sparse and device-specific memory vulnerabilities, achieving high attack success rates with minimal fault injections.

Next, we propose a hybrid approach for the detection of Spectre gadgets using Generative Adversarial Networks for generating diverse gadget datasets and a BERT-based classifier for high-dimensional analysis. This methodology significantly improves the scalability and comprehensiveness of gadget detection in large software systems.

Then, we explore the use of Large Language Models for patching source code vulnerabilities caused by microarchitectural side-channel leakages. By designing carefully structured prompts and leveraging dynamic analysis tools, we show that LLMs can generate efficient

and leakage-resilient patches, offering a scalable and cost-effective alternative to manual code mitigation.

Finally, we develop a Reinforcement Learning framework for the discovery of microarchitectural vulnerabilities. By simulating x86 instruction execution in a custom RL environment, we enable automated exploration of instruction sequences, uncovering novel transient execution mechanisms and previously unknown vulnerabilities.

Acknowledgments

The works that contributed to this dissertation was supported by the National Science Foundation, under grants CNS-1814406, CNS-2026913, CCF-2006738 and in part by Intel Corporation and Qatar National Research Fund.

I thank my advisor Professor Berk Sunar for his guidance and support throughout my Ph.D. studies and my committe members Professor Stjepan Picek and Professor Ziming Zhang for their guidance and feedback. I would also like to thank my co-authors Andrew J. Adiletta, Kemal Derya, Yarkın Doröz, Thomas Eisenbarth, Berk Gulmezoglu, Saad Islam, Koksal Mus, Kristi Rahman, and Koray Yurtseven for the work we have done together, and the anonymous reviewers for their valuable comments. Special thanks to Koksal Mus who encouraged me to do this Ph.D. and helped me start a life in the US. I am grateful to my family and friends for their support.

Contents

1	Introduction	1
1.0.1	Contributions	3
1.0.2	Publications	4
2	Background	7
2.1	Microarchitectural Vulnerabilities	7
2.1.1	Rowhammer Attacks	7
2.1.2	μ Arch Side Channel Attacks	8
2.1.2.1	Constant-Time Implementations	10
2.1.3	Transient Execution Attacks	11
2.1.4	Analysis Techniques for μ Arch Vulnerabilities	14
2.1.4.1	Detecting Side-channels	14
2.1.4.2	Detecting Spectre Gadgets	14
2.2	Machine Learning	16
2.2.1	Deep Neural Networks	16
2.2.2	Natural Language Processing	17
2.2.2.1	seq2seq Architecture	17
2.2.2.2	Generative Adversarial Networks	18
2.2.2.3	Attention-only Models	19
2.2.3	Reinforcement Learning	20

3	Search of Hardware Specific Fault Targets on Security-Sensitive Software	22
3.1	Motivation	22
3.2	Backdoor Attacks on DNN Models	24
3.3	Threat Model	25
3.4	Backdoor Injection using Rowhammer	27
3.4.1	Offline Attack Phase	27
3.4.1.1	Memory Profiling For Adjacent Rows	27
3.4.1.2	Memory Profiling For Faults	27
3.4.1.3	Constrained Fine-Tuning with Bit Reduction (CFT+BR) .	30
3.4.2	Online Attack Phase: Flipping Bits in the Deployed Model	34
3.4.2.1	Releasing the Flippy Rows	34
3.4.2.2	Mapping the Model Weights to Flippy Rows	35
3.4.2.3	Flipping Bits in the Weight File	35
3.4.3	Weight Quantization	36
3.5	Evaluation	37
3.5.1	Experimental Setup	37
3.5.2	Evaluation Metrics	38
3.5.3	Rowhammer Attack on Deployed Model - Online	39
3.5.4	CIFAR-10 Experiments	42
3.5.5	ImageNet Experiments	44
3.5.6	Generalization to Other DNN Architectures	46
3.6	Potential Countermeasures	46
3.6.1	Prevention-Based Countermeasures	47
3.6.2	Detection-Based Countermeasures	47
3.6.3	Recovery-based Countermeasures	49
3.7	Related Works	50
3.8	Discussion	52

3.9	Dynamic Analysis Approach on the Detection of Fault Targets	53
3.9.1	Tool Implementation	56
3.9.2	Experiments	57
3.9.2.1	ML Misclassification	57
3.9.2.2	Crypto Libraries	58
3.10	Conclusion	66
4	Scalable Generation and Detection of Spectre Gadgets	67
4.1	Motivation	67
4.2	Related Works	70
4.2.1	Spectre attacks and detectors	70
4.2.2	Binary Analysis with Embedding	71
4.2.3	GAN-based Text Generation	72
4.3	SpectreGAN: Spectre Gadget Generation	72
4.3.1	Gadget Generation via Fuzzing	72
4.3.2	SpectreGAN: Assembly Code Generation with GANs	75
4.3.2.1	SpectreGAN Architecture	75
4.3.2.2	Training	79
4.3.2.3	Tokenization and Training Parameters	80
4.3.2.4	Evaluation	81
4.3.3	Diversity and Quality Analysis of Generated Gadgets	83
4.3.3.1	Syntactic Analysis	83
4.3.3.2	Microarchitectural Analysis	85
4.3.3.3	Detection Analysis	86
4.4	FastSpec: Fast Gadget Detection Using BERT	89
4.4.1	Training Procedures	90
4.4.1.1	Pre-training	91
4.4.1.2	Fine-tuning	92

4.4.2	Training Details and Evaluation	92
4.4.3	Case Study: OpenSSL Analysis	93
4.4.4	Case Study: Phoronix Test Suite Analysis	94
4.5	Discussion and Limitations	99
4.5.1	Gadget Verification	99
4.5.2	Scalability and Flexibility	100
4.5.3	Comparison of FastSpec with Other Tools	101
4.5.4	Scope and Limitations	102
4.6	Conclusion	104
5	Automated Side-Channel Patching in Source Code Using LLMs	105
5.1	Motivation	105
5.2	Related Works	109
5.3	Threat Model and Scope	111
5.3.1	Research Questions	112
5.4	Methodology	112
5.4.1	Ensuring Constant-Time Execution	112
5.4.1.1	Evaluating Side-Channel Leakage	113
5.4.1.2	Patching for Constant-timeness	114
5.4.2	Mitigating Spectre-v1	119
5.4.2.1	Finding Spectre-v1 Gadgets	119
5.4.2.2	Patching Spectre-v1 Gadgets	120
5.5	Evaluation	121
5.5.1	Patching Spectre-v1 Gadgets	122
5.5.2	Patching a Real World Spectre-v1 Gadget	126
5.5.3	Patching Javascript Libraries for Constant-Timeness	126
5.5.4	Comparison of LLMs	129
5.6	Discussion and Limitations	132

5.7	Conclusion	134
6	Exploring μArch Vulnerabilities Using Reinforcement Learning	135
6.1	Motivation	135
6.2	Related Works	139
6.3	Threat Model and Scope	140
6.4	Our RL Framework	141
6.4.1	Environment	142
6.4.2	RL Agent	143
6.4.3	Action Space	143
6.4.4	State	144
6.4.5	Observation	144
6.4.6	Reward Function	146
6.5	Experiments	150
6.6	Discovered Transient Execution Mechanisms	151
6.6.1	Masked Exceptions	151
6.6.2	Transitions Between MMX and x87	152
6.7	Discussion	153
6.8	Conclusion	154
7	Conclusion	155
A	Spectre Gadget Generation	178
A.1	Assembly Gadget Examples	178
A.2	Mutational Fuzzing	180
B	Side-Channel Patching	181
B.1	Example Patching Loop with GPT-4	181
B.2	Microbenchmark of Leaky Functions Compiled from the Literature	183

C RL-based μArch Vulnerability Exploration	188
C.1 Instruction Sets	188

List of Tables

3.1	Average number of bit flips per memory page for DDR3 and DDR4 chips . . .	30
3.2	Comparison of our methods CFT, CFT+BR with the baseline methods Bad-Net, FT, and TBT	41
3.3	CFT+BR experiment results on VGG architectures	46
3.4	Number of gadget candidates found in decision tree algorithm with different Hamming distances.	58
3.5	Number of gadget candidates found by MFS on OpenSSL.	59
3.6	Ciphers implemented in OpenSSL that are vulnerable to LeapFrog attack. . .	61
3.7	Results from scans on the <code>liboqs</code> library, showing various issues encountered during signature operations for each digital signature scheme, along with the total number of assembly executions and candidate gadgets.	64
4.1	the number of unique n-grams for base gadgets and generated gadgets by fuzzing and SpectreGAN methods.	84
4.2	Comparison of oo7, Spectector, and FastSpec.	97
5.1	Parameter configurations of different LLMs used in this work.	122
5.2	Mitigation overhead of the Spectre-v1 micro benchmark for different mitigation techniques.	125
5.3	Patching performance of ZeroLeak in vulnerable Javascript libraries.	129
5.4	Patching performance of different models.	130

6.1 List of used CPU performance events available in Intel Core i9-7900X with descriptions	145
A.1 Instructions and registers inserted randomly in the fuzzing technique.	180
C.1 Number of instructions per set used in the action space	189

List of Figures

1.1	Overview of the dissertation. The subcategories of microarchitectural vulnerabilities and the aspects we address are listed in a tree structure. For each type of vulnerability, the corresponding publications are shown as leaf nodes. Excluded publications are indicated with dashed boxes.	6
2.1	An example of a data-dependent equality check logic that violates the constant-time property. Adapted from [87].	10
3.1	Backdoored Model behavior with clean inputs (top) and trigger added inputs (bottom). Fault injection to the model changes the behavior of the classifier, as shown by the confusion matrices.	25
3.2	The bit flip locations in one of the 4KB pages showing the sparsity of the bit flips.	29
3.3	The illustration of targeted model weights across the DNN model weight pages in the memory. The bulls-eye denotes the targeted bit location in a page. . .	33
3.4	Physical Address of released pages vs ResNet20 weight file. First pages of the weight file are mapped to the last released pages of our buffer.	36
3.5	Average number of bit flips on an 8MB buffer vs the number of sides in an n-sided Rowhammer attack.	40
3.6	Average number of bit flips per page for 15-sided (blue) and 7-sided (red) Rowhammer attack patterns.	40

3.7	Total loss graph at every training iteration during the backdoor injection to the ResNet18	42
3.8	The change in GradCAM [190] heatmaps that belong to ResNet18 before the attack (left) and after the attack (right). The focus of the model shifts through the trigger pattern if it is backdoored.	46
3.9	LeapFrog gadget detection using MFS framework	54
3.10	<code>aes-256-ctr</code> simulation results	60
3.11	<code>aria-256-ctr</code> simulation results. Plaintext <code>helloworld</code> is revealed three times.	60
3.12	LeapFrog gadget in <code>aria-128-cbc</code> block cipher.	61
3.13	LeapFrog gadget detected in <code>liboqs</code> binary for Dilithium PQC Digital Signature Scheme. The PC value that fault is injected into, $addr_{src}$, is highlighted in blue. The new value after the fault injected, $addr_{dest}$, is highlighted in red. The fault is injected during the execution of the function call highlighted in green.	65
4.1	SpectreGAN architecture.	76
4.2	The validation perplexity and Spectre gadget success rate for SpectreGAN. .	82
4.3	The distribution of base, fuzzing generated and SpectreGAN generated gadgets.	85
4.4	3-D visualization for the distribution of instructions and registers after t-SNE is applied to embedding vectors.	90
4.5	Solid line stands for the ROC curve of FastSpec for Spectre gadget class. Dashed line represents the reference line.	95
4.6	The processing time of FastSpec is independent of the number of branches whereas for Spectector and oo7 the analysis time increases drastically.	98
5.1	ZeroLeak patch generator framework overview.	114
5.2	Prompt template for constant time patch.	118
5.3	Prompt template for patching Spectre-v1 gadgets.	121

5.4	Spectre v1 patch examples on source code. The top one shows inline <code>lfence</code> mitigation. The bottom one shows the patch generated after our framework.	124
5.5	Patching OpenSSL Spectre gadget example	127
5.6	A failed example from <code>codechat-bison</code> . Original function is on top, and the generated patch is below.	131
6.1	Overview of the RL framework for μ Arch vulnerability analysis.	142
6.2	Test flow for detecting observable byte leakage.	149
6.3	Experiment Setup	150

Chapter 1

Introduction

In recent years, the increasing complexity and performance demands of modern computing systems have driven the development of microarchitectural optimizations. These optimizations—essential to improving computation speed and efficiency—come with unintended security risks. By exploiting subtle interactions within hardware components, memory access patterns, and speculative execution processes, attackers have discovered a range of novel vulnerabilities. Among these, Rowhammer and transient execution attacks have demonstrated the potential to compromise data integrity, confidentiality, and even control-flow integrity in a range of computing environments. This thesis explores how artificial intelligence (AI) can assist in identifying, analyzing, and mitigating these microarchitectural vulnerabilities, providing automated, scalable tools for security in modern systems. Our work further examines how these hardware-level vulnerabilities impact machine learning (ML) models, which are increasingly embedded in security-sensitive applications, presenting new challenges in robustness and resilience.

One of the prominent examples of fault attacks, Rowhammer, leverages repetitive access to specific DRAM rows to induce charge leakage in adjacent rows, resulting in bit flips [101]. Since its discovery, Rowhammer has been shown to exploit hardware weaknesses across diverse computing domains, from local devices to cloud and edge environments. Ef-

orts to mitigate Rowhammer have yielded a range of defenses, from software-based detection methods [39, 90] to hardware-level countermeasures. However, these solutions often fail to address Rowhammer’s growing sophistication fully. The discovery of Rowhammer gadgets [204], which leverage predictable memory accesses to execute unintended behavior, highlights Rowhammer’s potential to compromise even more secure systems. Similarly, recent work by Adiletta et al. [9] demonstrated that Rowhammer can target internal CPU states, creating new risks to stack variables and sensitive data.

The growing adoption of machine learning models in security-critical applications introduces additional complexity to the microarchitectural vulnerability landscape. Deep neural networks (DNNs), widely used for tasks like image classification, anomaly detection, and natural language processing, are vulnerable to adversarial and fault injection attacks that can compromise both model accuracy and integrity [66, 199]. Fault injection attacks, such as Rowhammer, can target ML models by flipping bits in their weights, resulting in significant accuracy degradation or even targeted misclassifications [81, 240].

Transient execution attacks, including Spectre and Meltdown, represent another significant class of microarchitectural vulnerabilities. These attacks exploit speculative and out-of-order execution, core mechanisms in modern CPUs designed to maximize performance. Spectre, one of the earliest and most widely publicized of these attacks, manipulates speculative execution paths to expose sensitive data [105]. By tricking the CPU into performing speculative operations on malicious data, attackers can infer confidential information through side-channel analysis. Meltdown, a closely related attack, leverages out-of-order execution to access unauthorized memory locations, creating further security risks in multi-user and multi-tenant environments [119]. As these vulnerabilities became widely known, they prompted extensive research into speculative execution defenses, but the effectiveness of current mitigations remains limited.

Addressing the Spectre and Meltdown vulnerabilities has proven challenging. Hardware patches are often infeasible due to the prohibitive costs associated with redesigning

CPUs, while software mitigations frequently degrade performance, limiting their appeal. Additionally, current automated detection tools, including taint analysis and symbolic execution [72, 224], struggle to scale effectively for large binaries and complex dependencies. Our work introduces an AI-driven tool that addresses these limitations by leveraging generative models, including GANs and Transformers, to automatically detect and mitigate speculative execution vulnerabilities at scale. This approach provides a more robust, flexible means of identifying and patching Spectre vulnerabilities, allowing developers to address microarchitectural risks without compromising performance.

This thesis investigates the application of AI techniques to improve the resilience of modern systems against microarchitectural vulnerabilities.

1.0.1 Contributions

In summary, this thesis makes the following contributions:

- We develop a novel *constrained optimization* algorithm that identifies memory bit locations vulnerable to Rowhammer and maps model weights to create backdoors effectively. Our optimization jointly minimizes the number of model modifications required for backdooring by simultaneously optimizing trigger patterns, vulnerable bit locations, and model parameter values. We validate the practicality of our approach by targeting a deployed ResNet-20 model trained on CIFAR-10 in PyTorch. With Rowhammer performed on live DRAM, the model retains over 91% test accuracy and achieves a 94% backdoor attack success rate by flipping only 10 out of 2.2 million bits. Through experiments, we show that state-of-the-art countermeasures against bit-flip attacks are either ineffective (e.g., weight reconstruction, piecewise weight clustering), introduce significant overhead (e.g., weight encoding), or degrade accuracy considerably (e.g., binarization-aware training) against our attack.
- We develop MFS, the first simulation tool designed to identify LeapFrog gadgets. Built on Intel Pin, it systematically analyzes binaries and incorporates time-domain analysis,

improving upon existing methodologies. Using MFS, we scan the Open Quantum Safe library, OpenSSL encryption, and a machine learning model to quantify potential LeapFrog gadgets in their codebases.

- We present the first comprehensive study of LLMs to automatically patch microarchitectural side-channel vulnerabilities. To the best of our knowledge, this is the first work to propose an automated method to fix side channels in the source code, which eases the shortage of developers with security expertise in the CI/CD pipeline.
- We propose prompting techniques and a toolchain leveraging LLMs to detect vulnerabilities, generate security patches, and evaluate performance and cost, demonstrating effectiveness across programming languages and real-world libraries.
- We propose a novel approach for discovering microarchitectural vulnerabilities using reinforcement learning (RL).
- We design a custom RL environment simulating the execution of x86 instructions on a microarchitecture, enabling the RL agent to explore the instruction space effectively.
- We show that the RL agents are able to discover unknown transient execution mechanisms, such as masked floating-point exceptions and MME/x87 transitions, showcasing its capability in identifying novel vulnerabilities.

1.0.2 Publications

The work presented in this dissertation was a collaborative effort with several co-authors. The overview of the chapters and their corresponding publications are shown in Figure 1.1. The following publications are associated with this dissertation:

1. **M. Caner Tol**, Saad Islam, Andrew J. Adiletta, Berk Sunar, and Ziming Zhang. "Don't Knock! Rowhammer at the Backdoor of DNN Models." In 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) IEEE, 2023.

2. **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) IEEE, 2021.
3. **M. Caner Tol**, and Berk Sunar. "ZeroLeak: Automated Side-Channel Patching in Source Code Using LLMs." In European Symposium on Research in Computer Security, pp. 290-310. Cham: Springer Nature Switzerland, 2024.
4. **M. Caner Tol**, Kemal Derya, and Berk Sunar. " μ RL: Discovering Transient Execution Vulnerabilities Using Reinforcement Learning" Preprint (2024).
5. Andrew J. Adiletta, **M. Caner Tol**, and Berk Sunar. "LeapFrog: The Rowhammer Instruction Skip Attack." hardwear.io (2024).

The following publications are excluded from this dissertation:

1. Koksal Mus, Yarkin Doröz, **M. Caner Tol**, Kristi Rahman and Berk Sunar, "Jolt: Recovering TLS Signing Keys via Rowhammer Faults," 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2023
2. Kemal Derya, **M. Caner Tol**, and Berk Sunar. "FAULT+PROBE: A Generic Rowhammer-based Bit Recovery Attack." Preprint (2024).
3. Berk Gulmezoglu, Andreas Zankl, **M. Caner Tol**, Saad Islam, Thomas Eisenbarth, and Berk Sunar. 2019. Undermining User Privacy on Mobile Devices Using AI. In Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19). Association for Computing Machinery, New York, NY, USA
4. Andrew Adiletta, **M. Caner Tol**, Yarkin Doröz, and Berk Sunar. 2024. Mayhem: Targeted Corruption of Register and Stack Variables. In Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIA CCS '24). Association for Computing Machinery, New York, NY, USA, 467–482

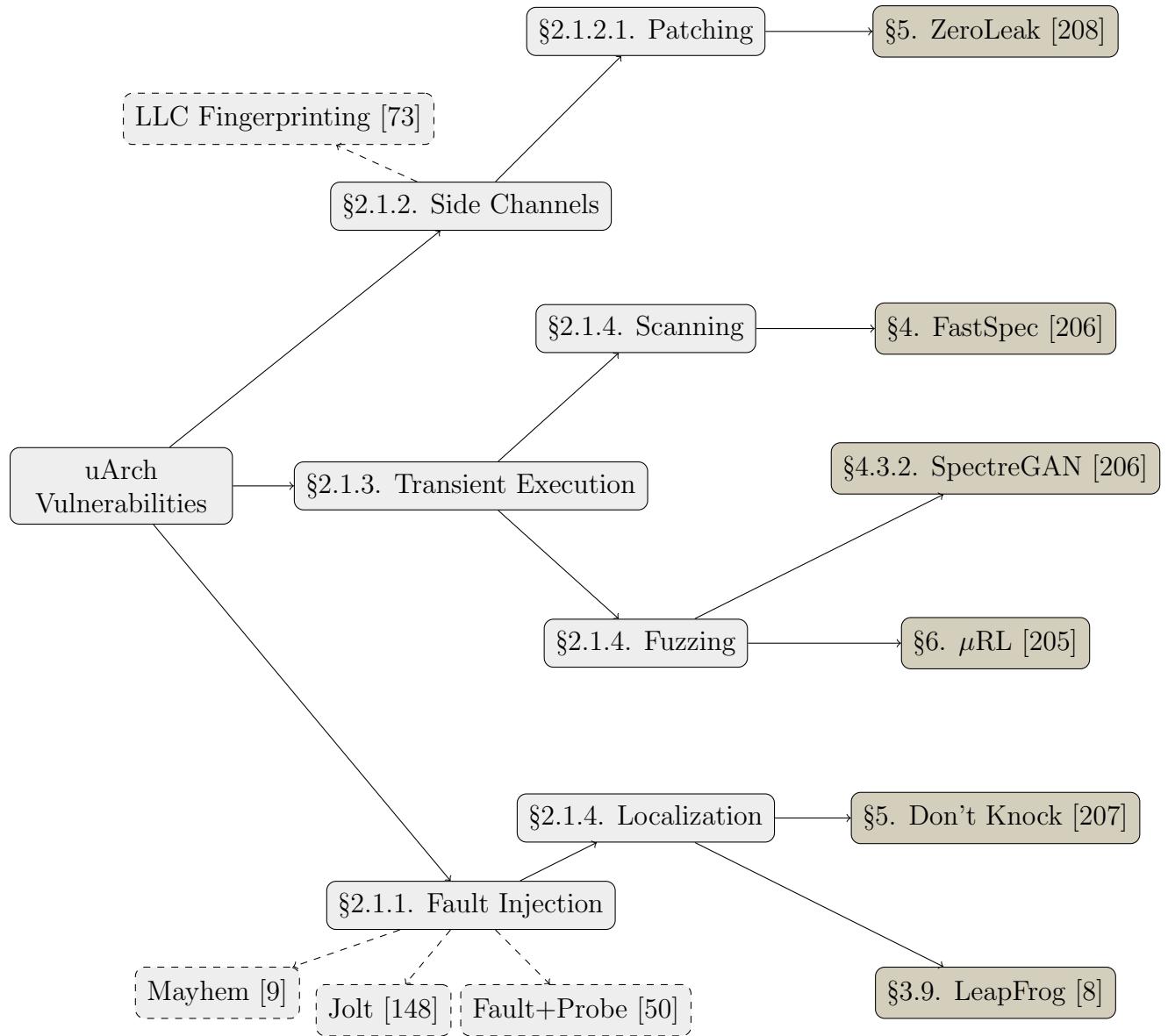


Figure 1.1: Overview of the dissertation. The subcategories of microarchitectural vulnerabilities and the aspects we address are listed in a tree structure. For each type of vulnerability, the corresponding publications are shown as leaf nodes. Excluded publications are indicated with dashed boxes.

Chapter 2

Background

2.1 Microarchitectural Vulnerabilities

2.1.1 Rowhammer Attacks

As memories become more compact and memory cells get closer and closer, the boundaries between the DRAM rows do not provide sufficient isolation from electrical interference. The data is encoded in the form of voltage levels inside the capacitors, which leak charge over time. Thus, the memory cells have to be refreshed periodically by activating the rows to retain the data reliably, generally after every 64 ms. Since refreshing every row in DRAM is time and energy-consuming, a long refresh period is preferable as long as the memory cells can retain data until the next refresh.

Kim et al. [101] identified that when the voltage of a row of memory cells is switched back and forth, nearby memory cells cannot retain the stored data until the next refresh, causing bit flips. Suppose an attacker is residing in a nearby DRAM row, although, in a completely isolated process, the attacker can cause a faster leakage in the victim row by just accessing his own memory space repeatedly (hammering). Since the Rowhammer vulnerability has been discovered, it was rigorously analyzed [100, 162] and many exploits, such as unauthorized access to a co-hosted VM [179], Android root exploit [218], and recovery of secret crypto

keys [55, 92, 148, 149], was shown. Recently, [49, 59, 94] have shown that more than 80% of the DRAM chips in the market are vulnerable to the Rowhammer attack including DDR4 chips having Target Row Refresh (TRR) mitigation. [78] proposed a methodology that results in bit flips in 99.9% of all DRAM rows on DDR4 chips with TRR protection. The Error Correcting Codes (ECC) mitigation has also been bypassed in [43]. Rowhammer is a significant threat to shared cloud environments [42, 239] as it can be launched across virtual machine (VM) boundaries and even remotely through JavaScript. Two research teams concurrently [120, 202] showed even a remote machine can induce Rowhammer bit flips by sending network packets. More recently, [170] have shown a combined effect of more than two aggressor rows to induce bit flips in recent generations of DRAM chips. All existing Rowhammer defenses including TRR, ECC, detection using Hardware Performance Counters, and changing the refresh rate can not fully prevent the Rowhammer attack [59, 68]. The only requirement of the Rowhammer attack is that the attacker and the victim share the same DRAM chip, vulnerable to the Rowhammer attack.

Terminal Brain Damage [81] attack showed that DNN model weights are vulnerable to Rowhammer since bit-flip corruptions can alter the value of floating-point numbers significantly, causing accuracy degradation and even targeted misclassification. Deephammer [240] showed that Rowhammer can deplete the accuracy of quantized DNN models as well.

2.1.2 μ Arch Side Channel Attacks

The state of the shared μ Arch resources, such as cache, DRAM, internal buffers and TLB, can be observed by a colocated attacker to infer the secret data of the victim. In this work, we focus on cache side-channels. Over the past years, different techniques have been developed to extract sensible data by using cache timing as a side-channel attack.

FLUSH+RELOAD [241] leverages the last-level cache (LLC) to monitor memory access patterns in shared pages. While it does not require the attacker and victim to share the same execution core, it flushes a potential victim address from the cache, and then measures

the reload time if the target address is accessed. EVICT+RELOAD [70] is another work where an eviction technique is used when cache flushing is not available.

Flush+Flush [69] attack introduces a novel method for exploiting cache timing vulnerabilities that relies solely on the execution time of the `clflush` instruction in x86 CPUs rather than the memory accesses, setting it apart from traditional cache attacks. Evict+Time [163] attack exploits the timing difference between cache hits and misses to infer cache state. By measuring execution time, an attacker can deduce if a cache miss occurred, revealing information about data access patterns. Evict+Spec+Time [38], a refined version of Evict+Time, allows attackers to determine not only the presence of a cache miss but also the exact location within the victim’s code where it occurred. The new method proves highly effective, significantly outperforming Evict+Time in efficiency.

Cache contention attacks fill a cache set and measure re-access time. Other processes using the set evict the attacker’s cache lines, causing higher latency, which reveals their cache activity. Prime+Probe [122] uses eviction sets to evict victim data from the cache instead of using the `clflush` instruction without requiring page deduplication between attacker and victim, making it more versatile across different environments.

The work [89] presents cross-core cache attack that leverages access time variations in the LLC to retrieve sensitive information across virtual machine (VM) boundaries. The attack exploits huge pages to function without memory deduplication, requiring only machine co-location of the attacker and victim on separate cores. PRIME+ABORT [53] attack introduces a novel approach to LLC attacks by eliminating the need for timing side channels, which traditional LLC attacks rely on. Instead, PRIME+ABORT uses Intel’s Transactional Synchronization Extensions (TSX), allowing it to bypass many existing defenses that target timer-dependent attacks. Prime+Scope [169] is a high-precision cross-core cache contention attack that enhances cache timing resolution by performing rapid, single-line cache contention measurements.

2.1.2.1 Constant-Time Implementations

Since many of the μ Arch design choices, such as hierarchical cache structures, it is not viable to eliminate the side-channel vulnerabilities entirely. CPU vendors, instead, publish software development guidelines to mitigate the timing side-channels in the software using constant-time programming techniques [87].

Constant-time implementations refer to cryptographic algorithms and methods that take a constant amount of time to execute, regardless of the input size or values. This type of implementation is essential for securing systems against timing attacks. A practical example of this could be seen in the RSA decryption algorithm, where different execution paths can be chosen based on the secret key bit. An attacker can utilize this timing discrepancy to infer the secret key [106]. The implementation process of constant-time cryptographic algorithms typically requires meticulous programming to ensure that no branches (such as if-then-else constructs), loops, or other operations are contingent on the secret data. For instance, cryptographic algorithms like AES should avoid data-dependent branches and employ bit-wise operations, which are known to execute in constant time on most platforms.

```
1 bool equals(byte a[], size_t a_len,
2             byte b[], size_t b_len) {
3     for (size_t i = 0; i < a_len; i++)
4         if (a[i] != b[i]) // data dependent!
5             return false;
6     return true;
7 }
```

Figure 2.1: An example of a data-dependent equality check logic that violates the constant-time property. Adapted from [87].

Challenges exist in guaranteeing a truly constant-time implementation, particularly on contemporary CPUs that possess features like out-of-order execution and speculative execution. This necessitates an in-depth understanding of both the cryptographic algorithm and the hardware it functions on.

There are several examples of constant-time cryptographic algorithms, such as the *constant-*

time carry-less multiplication utilized in AES-GCM implementations and the *constant-time modular inversion* employed in elliptic curve cryptography.

A plethora of tools exist for automated verification of the constant-time criterion. However, there is a significant discrepancy between academic research and cryptographic engineering practice. Despite the availability of tools for checking constant-time execution, developers often overlook this due to resource constraints [93].

Considering the escalating sophistication of side-channel attacks, the increasing heterogeneity, and the constant evolution of computing platforms, security-critical software needs to be continuously reevaluated for constant-time execution. Future research and developmental efforts will perpetually focus on generating more secure and efficient constant-time algorithms.

2.1.3 Transient Execution Attacks

In order to keep the pipeline occupied at all times, modern CPUs have sophisticated μ Arch optimizations to predict the control flow and data dependencies, where some instructions can be executed ahead of time. However, the predictions are not 100% accurate, causing them to execute some instructions mistakenly. These instructions cause pipeline flush once they are detected, and their results are never committed. Interestingly, μ Arch optimizations make it possible to leak secrets. The critical period before the flush is commonly referred to as the transient domain.

Transient execution attacks exploit speculative and out-of-order execution in CPUs to access secret data in the transient domain, leaving traces in the cache that attackers can analyze. Transient execution occurs when the CPU mispredicts the control flow or data dependencies [158], or when the executed instructions require μ code assist or cause exceptions [28, 174]. There are two classes of attacks in the transient domain: Spectre-type attacks that exploit the speculative execution and Meltdown-type attacks which exploit delayed permission checks and lazy pipeline flush in the re-order buffer [28].

Spectre Since a typical software consists of branches and instruction/data dependencies, modern CPUs have components for predicting conditional branches' outcomes to execute the instructions speculatively. These components are called branch prediction units (BPU), which use a history table and other components to make predictions on branch outcomes.

Spectre v1 [105], also known as *Bounds Check Bypass* or *Spectre-BHT*, affects a wide range of modern processors, including those from Intel, AMD, and ARM. It allows an attacker to trick a program into speculatively executing code that should not have been executed, potentially leaking sensitive data. In Spectre attacks, an attacker fills the branch history table with malicious entries such that the BPU makes a misprediction. Then, the CPU executes a set of instructions speculatively. As a result of misprediction, sensitive data can be leaked through μ Arch components, for instance, by encoding the secret to the cache lines to establish a covert channel.

```

1 void victim_function(size_t x){
2     if(x < size)
3         temp &= array2[array1[x] * 512];
4 }
```

Listing 2.1: Spectre-V1 C Code

For example, in the Spectre gadget in Listing 2.1, the 2nd line checks whether the user input **x** is in the bound of **array1**. In a normal execution environment, if the condition is satisfied, the program retrieves x^{th} element of **array1**, and a multiple of the retrieved value (512) is used as an index to access the data in **array2**. However, under some conditions, the **size** variable might not be present in the cache. In such occurrences, instead of waiting for **size** to be available, the CPU executes the next instructions speculatively. To eliminate unnecessary stalls in the pipeline. When **size** becomes available, the CPU checks whether it made a correct prediction or not. If the prediction was wrong, the CPU rolls back and executes the correct path. Although the results of speculatively executed instructions are not observable in architectural components, the access to the **array2** leaves a footprint in

the cache, making it possible to leak the data through side-channel analysis.

Spectre v1 is challenging to mitigate because it is a hardware-level issue, and traditional software-based security measures are not sufficient to fully protect against it. Since Spectre v1 is a complex vulnerability with widespread implications across different processor architectures and generations, it has been an ongoing challenge for the industry to address comprehensively.

NetSpectre [189] is the first remote variant of the Spectre attack, extending its reach beyond local code execution. NetSpectre marks a significant shift from local to remote attacks, making Spectre a threat even to systems where no attacker-controlled code is executed, including cloud environments.

SgxPectre attack [36] is a method of exploiting CPU vulnerabilities to compromise the confidentiality and integrity of SGX enclaves. By manipulating branch prediction from outside the enclave, attackers can temporarily alter the enclave’s control flow, producing cache-state changes that reveal sensitive information within the enclave.

Meltdown Meltdown [119] is an attack that bypasses memory isolation by exploiting out-of-order execution in modern processors to access protected kernel memory. This enables attackers to read memory from other processes or virtual machines without permission. Foreshadow [215] leaks SGX enclave secrets without needing kernel access or assumptions about enclave code. Rogue In-flight Data Load (RIDL) [219] leaks in-flight data directly from CPU *line fill buffers* without relying on cache or translation structures. Fallout [27] showed faulting loads caused by a non-present page fault can leak leftover values from the store buffer. ZombieLoad [187] uses μ code assists to transiently access data in the fill buffer. Load Value Injection (LVI) [217] injects attacker-controlled values into a victim’s transient execution. Downfall [141] exploits the `gather` instruction on AVX instruction set to leak data from the vector registers of the victim processes.

Some researchers proposed new designs requiring a change in the silicon level [99, 108]

while others proposed software solutions to mitigate transient execution attacks [165, 213]. Although these mitigations are effective against Spectre-type attacks, most of them are not used because of the drastic performance degradation [29] or the lack of support in the current hardware. Hence, Spectre-type attacks are not entirely resolved yet, and finding an efficient countermeasure is still an open problem.

2.1.4 Analysis Techniques for μ Arch Vulnerabilities

2.1.4.1 Detecting Side-channels

Microwalk-CI [232] is a dynamic side-channel analysis framework for easy integration into a JavaScript development workflow. Microwalk-CI adapts the existing Microwalk [231] framework, which was originally designed for finding leakages in binary software. For this, Microwalk generates a number of execution traces for a set of random inputs and then compares them using mutual information (MI), a robust measure that allows quantitatively assess the extent of information leakage. MI can capture a wide range of potential leakages, including those from the execution path and memory accesses. However, it is worth noting that *Microwalk* requires the tester to generate an input template file for each function tested and requires interpretation of the report file as it generates entropy estimates.

2.1.4.2 Detecting Spectre Gadgets

There are two main program analysis techniques that are commonly used to detect Spectre gadgets.

Taint Analysis: Taint analysis tracks outside user-controlled variables that possibly leak any secret data. If the tainted variables are consumed by a new variable in the program flow, the latter is also tainted in the information flow. This technique is commonly used in vulnerability detection [151], malware analysis , [19, 243] and web applications [18, 153] where user input misuses are highly likely. Similarly, in Spectre gadgets, the secret dependent operations after conditional branches are potential secret leakage sources. In particular, when

the branch decision depends on the user input, the secret is subject to be revealed in the speculative execution state. In order to detect the Spectre-V1 based leakage in benign programs, the taint analysis technique is used in *oo7* [225]. *oo7* employs control flow extraction, taint analysis, and address analysis to detect tainted conditional branches and their ability to impact memory accesses. *oo7* proposes selectively inserting a small number of fences instead of inserting fences after every conditional branch to minimize the overhead experienced by patching against Spectre. For instance, *oo7* reports less than 2% performance overhead in experiments on GNU Core utilities.

Symbolic Execution: Symbolic execution is a technique to analyze the program with symbolic inputs. Each path of the conditional branch is executed symbolically to determine the values, resulting in unexpected bugs. The symbolic execution is applied to detect potential information leakage in benign applications. For instance, *Spectector* [72] aims to identify the memory and control leaks by supplying symbolic inputs to target functions. They introduce the notion of *speculative non-interference* (SNI), and develop an algorithm based on symbolic execution to automatically prove SNI or detect violations indicating Spectre vulnerabilities which then can be patched. SNI requires that speculatively executed instructions do not leak more information into the microarchitectural state than what the intended behavior does, i.e., what is leaked by the standard, non-speculative semantics.

KLEESpectre [223] aims to model cache usage with symbolic execution to detect speculative interference, which is based on KLEE symbolic execution engine. KLEESpectre is evaluated on Kocher’s Spectre v1 variants [104] and on cryptographic libraries `libTomCrypt`, `Linux-tegra`, `openssl` and `hpn-ssh`.

Pitchfork [30] is a symbolic analysis tool that verifies that code is constant-time with respect to secret values such as encryption keys or message plaintexts. Pitchfork uses under-constrained symbolic execution augmented with dynamic taint tracking to verify constant-time execution. In particular, it uses a shadow memory to track secrets even as they are stored and loaded from memory. Pitchfork also allows the specification of function hooks.

This allows Pitchfork to verify a code at the protocol level while ignoring the implementation of crypto primitives. Pitchfork was used to verify that a large portion of Mozilla’s NSS cryptographic library is constant-time while also finding several constant-time vulnerabilities.

While the symbolic execution provides a good understanding of underlying bugs for different input values, it is challenging to apply for large-scale projects due to high resource demand.

2.2 Machine Learning

2.2.1 Deep Neural Networks

Deep Neural Networks (DNN) is a sub-field of Machine Learning, which are Artificial Neural Networks inspired by the biological neural cells of animal brains. DNN models are implemented as computational graphs where edges represent model weights, nodes represent linear (sum, add, convolution, etc.), and non-linear operations (sigmoid, softmax, relu, etc.). DNN models are formed by multiple layers of weight parameters where each layer learns a different level of abstraction of the features hierarchically [247].

DNN models can be broadly classified into two categories: generative and discriminative models. Generative models focus on learning the joint probability distribution between the input data and their labels, whereas discriminative models aim to learn the conditional probability distribution of the labels given the input data.

Discriminative models that are mostly trained in a supervised manner, i.e., with labeled data. Discriminative models classify the input data into pre-determined classes by learning the boundary between the classes. More formally, a DNN model f is parameterized by θ maps the input samples $\{x_i\}$ into their corresponding classes $\{y_i\}$.

Training The model parameters θ are optimized using the data pairs $\{x_i, y_i\}$ according to the following objective,

$$\min_{\theta} F(\theta) = \sum_i \left[\ell(f(x_i, \theta), y_i) \right],$$

where F is the objective function, ℓ is a loss function, $\Delta\theta$ is the change in the model weights. The model is updated by backpropagating the errors through the layers [183]. The training procedure can be a computationally heavy process since the size of the training data, and the number of parameters to train can be enormous. Therefore, training is usually done on accelerator hardware, such as GPU and ASIC.

Inference After the model weights reach an acceptable performance on the training data set, they can be deployed as a part of the service. In the inference stage, the model weights are usually kept unchanged, and the model's output is used as the classifier output. Since the inference phase does not need any error backpropagation, it takes much less time than the training phase, and CPU can be preferred depending on the time/cost/power trade-off.

2.2.2 Natural Language Processing

2.2.2.1 seq2seq Architecture

Sequence to sequence mapping is a challenging process since the text data set has no numeric values. First, the text data is converted to numeric values with embedding methods [138, 140]. Then, a DNN model is trained with vector representations of the text.

A new approach called seq2seq [196] was introduced to model sequence-to-sequence relations. The seq2seq architecture consists of encoder and decoder units. Both units leverage multi-layer Long Short Term Memory (LSTM) structures where the encoder produces a fixed dimension encoder vector. The encoder vector represents the information learned from the input sequence. Then, the decoder unit is fed with the encoder vector to predict the input sequence's mapping sequence. After the end of the sequence token is produced by the decoder, the prediction phase stops. The seq2seq structure is commonly used in chatbot

engine [171] since sequences with different lengths can be mapped to each other.

2.2.2.2 Generative Adversarial Networks

A specialized method of training generative models was proposed by Goodfellow et al. , [65] called generative adversarial networks (GANs). The generative models are trained with a separate discriminator model under an adversarial setting. In [65], the training of the generative model is defined as,

$$\begin{aligned} \min_G \max_D V(D, G) = & \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] \\ & + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \end{aligned} \quad (2.1)$$

In Equation 2.1, the generator G and the discriminator D are trained in such a way that D , as a regular binary classifier, tries to maximize its confidence $D(x)$ on real data x , while minimizing $D(G(z))$ on generated samples by the G . At the same time, G tries to maximize the confidence of discriminator $D(G(z))$ on generated samples $G(z)$ and minimize $D(x)$ where x is the real data.

MaskGAN [56] is a type of conditional GAN technique to establish a good performance out of traditional GANs. MaskGAN is based on seq2seq architecture with an attention mechanism which improves the performance of the fixed-length encoder vectors. Each time a prediction is made by the decoder unit, a part of the input sequence is used instead of the encoder vector. Hence, each token in the input sequence has a different weight on the decoder output. The main difference of MaskGAN from other GAN-based text generation techniques is the token masking approach, which helps to learn the missing texts in a sequence. For this purpose, some tokens are masked that is conditioned on the surrounding context. This technique increases the chance of generating longer and more meaningful sequences out of GANs.

2.2.2.3 Attention-only Models

Although recurrent models with attention mechanisms learn the representations of long sequences, attention-only models, namely *Transformer* architectures [220], are shown to be highly effective in terms of computational complexity and performance on long-range dependencies. Similar to *seq2seq* architecture, the *Transformer* architecture consists of an encoder-decoder model. The main difference of *Transformer* is that recurrent models are not used in encoder or decoder units. Instead, the encoder unit is composed of L hidden layers where each layer has a multi-head self-attention mechanism with A attention heads and a fully connected feed-forward network. The input embedding vectors are fed into the multi-head attention, and the output of the encoder stack is formed by a feed-forward network, which takes the output of the attention sub-layer. The decoder unit also has L hidden layers, and it has the same sub-layers as the encoder. In addition to one multi-head attention unit and one feed-forward network, the decoder unit has an extra multi-head attention layer that processes the encoder stack output. To process the information in the sequence order, positional embeddings are used with token embeddings where both embedding vectors have a size of H .

Keeping the same Transformer architecture, Devlin et al. [51] introduced a new language representation model called BERT (Bidirectional Encoder Representations from Transformers), which surpasses the state-of-the-art scores on language representation learning. BERT is designed to pre-train the token representation vectors of deep bidirectional Transformers. For a detailed description of the architecture, we refer the readers to [51, 220]. The heavy part of the training is handled by processing unlabeled data in an unsupervised manner. The unsupervised phase is called *pre-training*, which consists of masked language model training and next sentence prediction procedures. The supervised phase is referred to as *fine-tuning*, where the model representations are further trained with labeled data for a text classification task. Both phases are further explained in detail for Spectre gadget detection model in Section 4.4.

2.2.3 Reinforcement Learning

In RL, the objective is for an agent to learn a policy $\pi_\theta(a|s)$, parameterized by θ , which maximizes the expected cumulative reward through its chosen actions in an environment. The policy gradient method [198] computes the gradient of the expected reward with respect to the policy parameters, allowing the agent to directly update the policy by following the gradient. Formally, the objective function $J(\theta)$ can be defined as:

$$J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^T r_t \right], \quad (2.2)$$

where r_t is the reward at time step t , and the expectation is over the trajectories induced by the policy π_θ . The policy is updated by adjusting θ in the direction of the gradient $\nabla_\theta J(\theta)$ using gradient ascent.

One of the major challenges with vanilla policy gradient methods is the high variance of the gradient estimates, which can lead to unstable learning. Additionally, large updates to the policy parameters θ can cause dramatic changes to the policy, potentially leading to performance collapse.

Trust Region Policy Optimization (TRPO) [185] was proposed to address this issue by enforcing a constraint on the size of policy updates using a trust region. TRPO introduces the following constrained optimization problem:

$$\max_{\theta} \mathbb{E}_{\pi_\theta} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \hat{A}(s, a) \right] \text{ subject to } \mathbb{E}_s [D_{\text{KL}} (\pi_{\theta_{\text{old}}} \| \pi_\theta)] \leq \delta, \quad (2.3)$$

where D_{KL} is the Kullback-Leibler (KL) divergence, $\hat{A}(s, a)$ is the advantage estimate, and δ is a small positive value controlling the step size. However, TRPO is computationally expensive due to the need for second-order optimization to enforce the KL-divergence constraint.

Proximal Policy Optimization (PPO) [186] simplifies TRPO by replacing the hard constraint on policy updates with a penalty or by using a clipped objective function. The key

idea behind PPO is to ensure that policy updates are “proximal” to the current policy, preventing drastic updates that could lead to instability.

In this work, we use PPO with *clipped objective*. In this approach, PPO clips the probability ratio $\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ to lie within a small interval around 1, preventing large updates. The clipped objective is defined as:

$$L^{\text{CLIP}}(\theta) = \mathbb{E} \left[\min \left(r(\theta) \hat{A}(s, a), \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}(s, a) \right) \right], \quad (2.4)$$

where $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ is the probability ratio, and ϵ is a small hyperparameter that limits how far the policy is allowed to change. By clipping the probability ratio, PPO discourages overly large updates while still allowing for sufficient exploration of the policy space.

Chapter 3

Search of Hardware Specific Fault Targets on Security-Sensitive Software

3.1 Motivation

DNN models are known for their powerful feature extraction, representation, and classification capabilities. However, the large number of parameters and the need for a large training data set make it hard to interpret the behavior of these models. The fact that an increasing number of security-critical systems rely on DNN models in real-world deployments raises numerous robustness and security questions. Indeed, DNN models have been shown to be vulnerable against imperceptible perturbations to input samples which can be misclassified by manipulating the network weights [66, 152, 199].

Emboldened by recent physical fault injection attacks such as Rowhammer, an alternative approach was proposed that directly targets the model when it is loaded into memory. There are two advantages of this attack:

1. Alternative approaches assume modifications are introduced to the model, either during distribution as part of a repository or after installation. Such malicious tampering may be challenging to implement in practice and can easily be detected.

2. In contrast, a Rowhammer-based attack can remain *stealthy* since the model is only modified in real-time while running in memory, and no input modification is required. Once the program is unloaded from memory, no trace of the attack remains except misclassified outputs.

Recently, [81, 240] showed that flipping a few bits in DNN model weights in memory while succeeding in achieving misclassification has the side-effect of significantly reducing the accuracy. Other works [16, 125] addressed this problem by tweaking only a minimum number of model weights that makes a DNN model misclassify a chosen input to a target label. This approach indeed achieves the objective with only a slight drop in classification accuracy.

Nevertheless, whether a practical attack such as injecting a backdoor to DNNs can indeed be achieved in a realistic and stealthy manner using Rowhammer in hardware is still an open question. Earlier approaches assume that Rowhammer can flip bits with perfect precision in the memory. This is far from what we observe in reality: only a small fraction of the memory cells are vulnerable; see Section 3.4.1.2 for further details. Therefore existing proposals fall short of presenting a practical DNN backdoor injection attack using Rowhammer. This motivates us to reconsider the backdoor injection process under new constraints, including the training algorithms.

Our contributions: In this work, we present a backdoor injection attack on a deployed DNN model using Rowhammer. This result shows that, indeed, real-life deployments are under threat from backdoor injection attacks. More work needs to be done to secure deployed models from fault injection attacks used for everyday tasks by end-users. More specifically,

- for the first time, we present an end-to-end backdoor injection attack realized on actual hardware on a classifier model using Rowhammer as the fault injection method
- we thoroughly characterize DRAMs for bit-flips using extensive Rowhammmer experiments. Our results show that previously proposed backdoor injection techniques make overly optimistic assumptions about Rowhammer,

- introduce a more realistic Rowhammer fault model, along with new stringent constraints on model modifications necessary to achieve a real-life attack,
- propose a novel *constrained optimization*-based algorithm that can map model weights to identify vulnerable bit locations in the memory to create a backdoor,
- we further reduce the number of modifications for the backdoor by jointly optimizing for trigger patterns, vulnerable locations, and model parameter values.
- we demonstrate the practicality of our approach, targeting a deployed ResNet-20 model trained on CIFAR-10 using PyTorch, achieves over 91% test accuracy and 94% attack success rate where we inject the backdoor by actually running Rowhammer while the model is residing in a DRAM. This high level of accuracy is reached by flipping only 10 out of 2.2 million bits.
- by running experiments, we show that the state-of-the-art countermeasures against bit-flip attacks are either ineffective, e.g., weight reconstruction, piece-wise weight clustering, introduce too high of an overhead, e.g., weight encoding, or significantly reduce the accuracy, e.g., binarization-aware training, to defend against our attack.

3.2 Backdoor Attacks on DNN Models

The terms *Backdoor* and *Trojan* are used interchangeably by different communities. Here we use Backdoor for consistency. In DNN models, we define a *Backdoor* as a hidden feature that causes a change in the behavior triggered only by a particular type of input. In the literature, backdooring is applied with either benevolent intents, such as watermarking the DNN models [7, 191], or with malicious purposes [14, 16, 37, 41, 71, 126], as a *Trojan* to attack the models.

In this dissertation, we focus on *Backdoor* as a type of *Trojan* exploited by an attacker to cause targeted misclassification. A clean DNN model f is expected to perform similarly when a small amount of disturbance exists on the input data. Therefore, $f(x_i + \Delta x, \theta) = y_i$

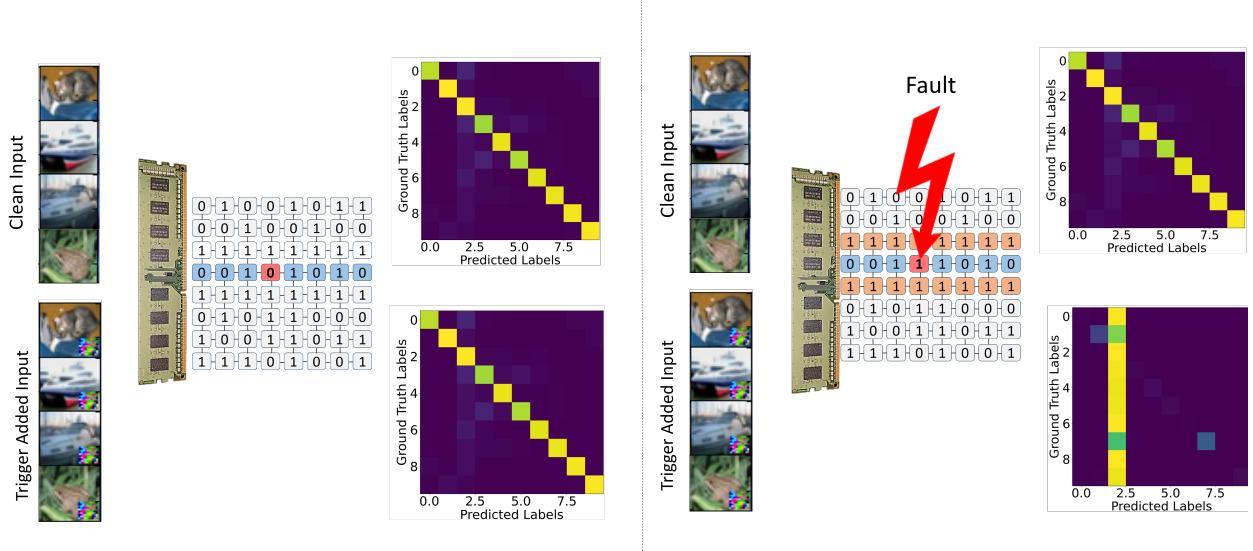


Figure 3.1: Backdoored Model behavior with clean inputs (top) and trigger added inputs (bottom). Fault injection to the model changes the behavior of the classifier, as shown by the confusion matrices.

if and only if $f(x_i, \theta) = y_i$, where Δx is a small disturbance on the input x . We say a DNN model f has a *backdoor* if $f(x_i, \theta) = y_i$ and $f(x_i + \Delta x, \theta) = \tilde{y}$.

Earlier works [14, 41, 71, 126] demonstrated that backdoor attacks pose a threat to the DNN model supply chain. Specifically, DNN models can be *backdoored* during the training phase if the model training is wholly or partially (transfer learning) outsourced [71]. Moreover, compromised model-training code can be an attack vector for backdoor attacks since it can train a backdoored model even if the model is trained with the local resources and clean training data set [14].

3.3 Threat Model

Same as in earlier works [71, 81, 126, 176, 240], we assume that the attacker

- knows the model architecture, parameters and the task of the target model;
- does not have access to the training hyperparameters or the training data set;
- has a small percentage of the unseen test data set;

- is involved only after the model deployment in a cloud server and therefore does not need to modify the software and hardware supply chain;
- resides in the same physical memory as the target model;
- has no more than regular user privileges (no root access).

Such threat models are well motivated in shared cloud instances targeting a co-located host running the model and in sandboxed browsers targeting a model residing in the memory of the host machine [42, 49, 239]. Moreover, the previous research on model stealing attacks [47, 95, 164, 212, 244] validates our white-box attack assumption. The test data required by our attack does not belong to the victim and is not in the training data set. Hence, it can be easily collected and labeled by the attacker since the task of the target model is known.

To better understand our attack, we illustrate an example in Figure 3.1. The attack works as follows:

1. *Offline Phase - Profiling Target Model and Memory:* By studying the model parameters and the memory, the attacker generates a trigger pattern and determines the vulnerable bits in the target model.
2. *Online Phase - Rowhammer Attack:* After the target model is loaded into the memory, using Rowhammer, the attacker flips the target bits by only accessing its own data that resides in the neighboring rows of the weight matrices in the DRAM.
3. *Targeted Misclassification:* After the backdoor is inserted, the model will misclassify trigger-added input to the target class. The misclassification will persist until the backdoored model is unloaded from the memory. Since the model in persistent storage (or in the software distribution chain) is untouched, malicious modification to the model is harder to detect.

3.4 Backdoor Injection using Rowhammer

3.4.1 Offline Attack Phase

In the offline phase of the attack, we optimize the trigger pattern and the bit-flip locations in the weight matrices. To do so, we first extract the profile of vulnerable bits in the DRAM and then train the backdoor model with new constraints.

3.4.1.1 Memory Profiling For Adjacent Rows

For the Rowhammer attack to work, we need to locate physical rows adjacent to victim rows that require finding physically contiguous memory addresses. We exploit SPOILER vulnerability [91] in Intel processors to determine which virtual addresses within an array are contiguous physically.

After performing SPOILER and determining which addresses are contiguous physically, these addresses need to be filtered even further to addresses that are within the same bank. This is again performed using another timing side-channel attack known as row conflict [167], which measures the difference in read times between two addresses to determine if the row buffer for the bank was cleared, resulting in a longer read time and extrapolating bank continuity.

3.4.1.2 Memory Profiling For Faults

Memory profiling is a process of finding vulnerable addresses in the DRAM. This process can be performed before the victim starts running. For DDR3 DRAMs, we implement a double-sided Rowhammer attack where we place a victim row between two attacker-owned rows. We set the victim rows to all zero and attacker rows to all one and repeatedly access the attacker rows. Then we check if there is any *zero to one* flip in the victim row. We find the *one to zero* flips similarly. For DDR4 systems, double-sided Rowhammer does not work due to the TRR mitigation implemented by the DRAM vendors. Therefore, we designate

alternating rows to be attacker and victim.

Assuming the bit flips are uniformly distributed over a memory page and a faulty memory cell can be flipped only in one direction, given a chain of bit offset $\{b_0, b_1, \dots, b_{k+l-1}\}$ in a memory page, the conditional probability of finding a suitable target page t in N flippy pages can be calculated as

$$p(t | \{b_{n_{0 \rightarrow 1}}\} \in \{0 \rightarrow 1\}, \{b_{n_{1 \rightarrow 0}}\} \in \{1 \rightarrow 0\}) = 1 - \left(1 - \prod_{i=0}^{k-1} \frac{n_{0 \rightarrow 1} - i}{S - i} \times \prod_{j=0}^{l-1} \frac{n_{1 \rightarrow 0} - j}{S - k - j}\right)^N, \quad (3.1)$$

where $n_{0 \rightarrow 1}$ and $n_{1 \rightarrow 0}$ are the average numbers of faulty memory cells in a page, flippable in the direction from 0 to 1 and 1 to 0 respectively, which are device-dependent values, k and l are number of bit locations which need to be flipped in the direction from 0 to 1 and 1 to 0 respectively, and S is the total number of bits in a page. Previous research [149] shows that $n_{0 \rightarrow 1}$ and $n_{1 \rightarrow 0}$ are almost equal to each other. Therefore, Equation 3.1 can be reduced as,

$$p(t | \{b_{n_{0 \rightarrow 1}}\} \in \{0 \rightarrow 1\}, \{b_{n_{1 \rightarrow 0}}\} \in \{1 \rightarrow 0\}) \approx 1 - \left(1 - \prod_{i=0}^{k+l-1} \frac{n_{0 \rightarrow 1} + n_{1 \rightarrow 0} - i}{S - i}\right)^N. \quad (3.2)$$

It takes 94 minutes to profile 128MB of memory, but this is done offline before the victim starts running. Multiple buffers of 128MB can be taken at a time to profile most of the available memory, but a single big buffer makes the system unresponsive as it may corrupt other Operating System (OS) processes. Figure 3.2 shows the sparsity of the bit flips in the profiled 128MB buffer and one of the 4KB pages in DDR3 and DDR4 DRAM chips.

Although we use state-of-the-art memory hammering techniques, we have found 34 bit flips in a 4KB page in DDR3. Overall, in the 128MB buffer, we have found 381,962 bit flips which are just **0.036%** of the total cells in the buffer, as illustrated in Figure 3.2. For profiling DDR4, we use a 15-sided Rowhammer attack. We tested 6 different DDR4 chips and averaged the number bit flips per page for each device. We also calculated the average

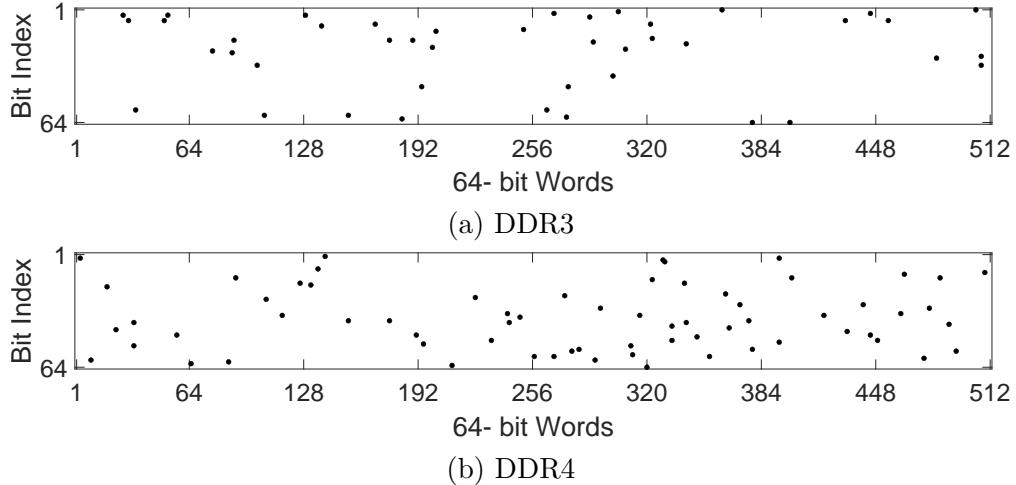


Figure 3.2: The bit flip locations in the profiled 128MB memory buffer and one of the 4KB pages show the sparsity of the bit flips. Only about 0.036% of the DRAM cells in the profiled memory are found to be vulnerable.

number of bit flips per page for the memory profiles published by earlier work [201] and summarized the results in Table 3.1.

Specifically, we can estimate the probability of finding a suitable target page by fixing the DRAM-specific parameter $n_{0 \rightarrow 1}$ and $n_{1 \rightarrow 0}$ for a DRAM using Equation 3.2. In line with the previous research [149] we also observe that number of bit flips from 0 to 1 and 1 to 0 are almost equal. Therefore, using the results of our profiling experiments, we estimate that $n_{0 \rightarrow 1} + n_{1 \rightarrow 0} = 34$. Total number of bits in a page is $S = 32,768$, and the total number of pages is $N = 32,768$ in a 128MB memory buffer where the page size is 4KB. Therefore, when $k = 1$, i.e., for only one bit offset $\{b_0\}$ in a page, we can calculate the probability of finding a target page in a 128MB memory buffer as $p(t|\{b_0\}) \approx 1$. Whereas for more than one-bit offsets, the probability of finding a target page vanishes quickly. Specifically, for $\{b_0, b_1\}$, $p(t|\{b_0\}) = 0.03$ and for $p(t|\{b_0, b_1, b_2\}) = 0.00003$. Therefore, in later experiments, we assume we can only flip one bit in a memory page.

Table 3.1: Average number of bit flips per memory page for 14 DDR3 and 6 DDR4 chips. The tags in DRAM columns represent different brand/model information. The results for DDR3 results are calculated from double-sided Rowhammer profiles [201]. DDR4 results are from the chips we profiled using n-sided Rowhammer.

	DRAM	Average # of Flips Per Page	DRAM	Average # of Flips Per Page
DDR3	A1	12.48	E1	12.46
	A2	1.92	E2	2.02
	A3	1.11	F1	28.77
	A4	15.85	G1	1.62
	B1	1.05	H1	1.66
	C1	1.60	I1	8.28
	D1	1.08	J1	1.25
DDR4	K1	100.68	L2	13.98
	K2	109.48	M1	2.04
	L1	3.12	N1	2.72

3.4.1.3 Constrained Fine-Tuning with Bit Reduction (CFT+BR)

We propose a novel *joint* learning framework based on constrained optimization to learn the bit flip pattern on the network weights as well as the data trigger pattern simultaneously. Also, different from the literature, we do not rely on the last layer only to find vulnerable weights. Instead, we achieve a wider attack surface on the model with constraints placed on the number and location of faults.

To preserve the performance on clean data, given a collection of test samples $\{x_i\}$ and their corresponding class labels $\{y_i\}$, we propose optimizing the following objective:

$$\begin{aligned} \min_{\Delta\theta \in \Delta\Theta} \max_{\|\Delta x\|_\infty \leq \epsilon} F(\Delta\theta, \Delta x) = \\ \sum_i \left[(1 - \alpha) \cdot \ell(f(x_i, \theta + \Delta\theta), y_i) + \alpha \cdot \ell(f(x_i + \Delta x, \theta + \Delta\theta), \tilde{y}) \right], \quad (3.3) \end{aligned}$$

where $\Delta\theta$, Δx denote the weight modification pattern and the data trigger pattern, \tilde{y} denotes the target label, ℓ denotes a loss function, f denotes the network parameterized by θ originally, $\alpha \in [0, 1]$ denotes a predefined trade-off parameter to balance the losses on clean

Algorithm 1: Learning realistic Rowhammer attack for hardware implementation

Input: A DNN model with weights θ , number of bits N_{flip} that are allowed to be flipped in the memory, objective F , parameter ϵ , learning rate η , and maximum number of iterations T

Output: Backdoored model θ^* and trigger pattern Δx^*

```
1  $\Delta\theta^* \leftarrow \emptyset, \Delta x^* \leftarrow \emptyset;$ 
2 for  $t \in [T]$  do
3   if update the trigger == true then
4      $\Delta x^* \leftarrow \Delta x^* + \epsilon \cdot \text{sgn}(\nabla_{\Delta x} F(\Delta\theta^*, \Delta x^*));$ 
5   end
6    $\mathcal{M} \leftarrow \text{Group\_Sort\_Select}(|\nabla_{\Delta\theta} F(\Delta\theta^*, \Delta x^*)|,$ 
7    $N_{flip}, \text{'descending'});$ 
8    $\Delta\theta^* \leftarrow \Delta\theta^* - \eta \cdot [\nabla_{\Delta\theta} F(\Delta\theta^*, \Delta x^*)]_{\mathcal{M}};$ 
9   if bit reduction == true then
10     $\theta^* \leftarrow \text{Floor}((\theta + \Delta\theta^*) \oplus \theta) \oplus \theta;$ 
11  end
12 end
13 return  $\theta^*, \Delta x^*$ 
```

data and triggered data. A large α value would cause the attack to give a more aggressive effort to increase the Attack Success Rate while sacrificing Test Accuracy, and a low α value would cause the attack to preserve the Test Accuracy while sacrificing the Attack Success Rate. Ideally, a moderate α value should be chosen to get a high Attack Success Rate while preserving the Test Accuracy as much as possible. Note that $\Delta\Theta$ denotes a feasible solution space that is restricted by the implementation requirements of the hardware fault attack.

Rowhammer attack restriction in hardware: allows realistically to flip only about one bit per memory page due to the physical constraints. Since the potentially vulnerable memory cells in the DRAM are sparse, the probability of finding a suitable target page to locate the victim is very low for more than one bit flip offsets (See Section 3.4.1.2). Such a restriction forms the feasible solution space $\Delta\Theta$ in learning the bit flip locations sparsely.

To solve the constrained optimization problem defined in Equation 3.3, we also propose a novel learning algorithm as listed in Algorithm 1 that consists of the following four steps:

Step 1. Learning data trigger pattern Δx : The goal of this step is to learn a trigger that can activate the neurons related to the target label \tilde{y} to fool the network. Trigger

pattern generation starts with an initial trigger mask. Then, we use the Fast Gradient Sign Method (FGSM) [66] to learn the trigger pattern. The update rule is defined as

$$\Delta x = \Delta x^* + \epsilon \cdot \text{sgn}(\nabla_{\Delta x} F(\Delta \theta^*, \Delta x^*)), \quad (3.4)$$

where $\Delta \theta^*$, Δx^* denote the current solutions for the two variables, ∇ denotes the gradient operator, and sgn denotes the signum function. $\epsilon \geq 0$ denotes another predefined parameter to control the trigger pattern. Since it acts as a learning rate of the trigger, smaller values update the trigger slower but may be more effective in finding the optimal pattern.

Step 2. Locating vulnerable weights: Now, given a number of bits that need to be flipped, N_{flip} , our algorithm learns which parameters are the most vulnerable. In this step, we apply two constraints to the optimization:

- C1. Locating one weight per bit flip towards minimizing our objective in Equation 3.3 significantly;
- C2. No co-occurrence in the same memory page among the flipped bits.

Recall that when a DNN model is fed into the memory, the network weights are loaded sequentially page-by-page, where each page is fixed-length and stored contiguously. We can view this procedure as loading a long vector by vectorizing the model. Therefore, to guarantee we choose at most one weight per memory page, we divide the network weight vector into N_{flip} groups as equally as possible, as illustrated in Figure 3.3. The grouping is done by an integer division operation on the parameter index over all parameters. If the index of a parameter is i_w , the group ID of that parameter is determined as $i_w \text{div} (4096 * N_{group})$ where N_{group} is the number of pages per bit flip, and **div** is integer division operation. N_{group} depends on the chosen number of bit flips N_{flip} and can be calculated as $N_{group} = N_w \text{div} (4096 * N_{flip})$ for a DNN model with number of parameters, N_w . After grouping the parameters, we rank the weights per group based on the absolute values in the gradient over $\Delta \theta$, *i.e.*, $|\nabla_{\Delta \theta} F|$ where $|\cdot|$ denotes the entry-wise absolute operator, in descending order.

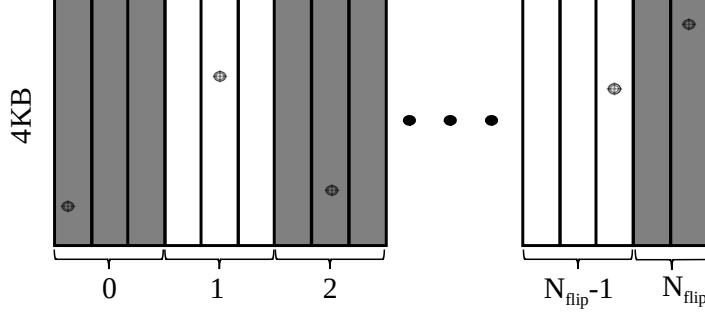


Figure 3.3: The illustration of targeted model weights across the DNN model weight pages in the memory. The bulls-eye denotes the targeted bit location in a page.

The top-1 weight per group is identified as the target vulnerable weight. Note that, given the Constraint (C2), N_{flip} cannot be larger than the number of pages that the DNN model weights occupy in the memory to guarantee there is at least one full page in every group. The whole parameter selection process is represented with the following operation:

$$\mathcal{M} \leftarrow Group_Sort_Select(|\nabla_{\Delta\theta} F(\Delta\theta^*, \Delta x^*)|, N_{flip}, 'descending'), \quad (3.5)$$

Step 3. Adversarial fine-tuning Now, given a collection of located vulnerable weights, denoted by \mathcal{M} , we only need to update these weights in backpropagation as follows:

$$\Delta\theta = \Delta\theta^* - \eta \cdot [\nabla_{\Delta\theta} F(\Delta\theta^*, \Delta x^*)]_{\mathcal{M}}, \quad (3.6)$$

where $[\cdot]_{\mathcal{M}}$ denotes a masking function that returns the gradients for the weights in \mathcal{M} , otherwise 0's, and $\eta \geq 0$ denotes a learning rate.

Step 4. Bit reduction To meet the physical constraints of the Rowhammer, the final part of our attack procedure requires bit reduction. Rowhammer can only flip a very low number of bits in a 4KB memory page, and more than one faulty memory cell almost never coexists within a byte. Therefore, we define a bit reduction function as $\text{Floor}(\theta \oplus \theta^*)$, where \oplus denotes the bit-wise summation, and function Floor rounds down the number by keeping the most significant nonzero bit only. For instance, letting $\theta = 1101_2$ and $\theta^* = 1010_2$, then

$\text{Floor}(\theta \oplus \theta^*) = \text{Floor}(0111_2) = 100_2$. In this way, we ensure that only one bit is modified in a selected weight while maintaining its change direction and amount as much as possible.

3.4.2 Online Attack Phase: Flipping Bits in the Deployed Model

When we access a file from the secondary storage, it is first loaded into the DRAM and when we close the file, the OS does not delete the file from DRAM to make the subsequent access faster. If the file is modified, the OS sets the dirty bit of that modified page and writes back according to the configured policy. Otherwise, the file remains cached unless evicted by some other process or file. As Rowhammer is capable of flipping bits in DRAM, we can use it in the online attack phase to flip the weights of the DNN file as it is loaded in the page cache. The weight file is divided into pages and stored in the page cache. We can flip our target bits as identified by the backdoored parameters θ^* , in Section 3.4.1. The OS does not detect this change as it is directly made in hardware by a completely isolated process, and it keeps providing the page cached modified copy to the victim on subsequent accesses. Thus, the attack remains stealthy. In the online phase, we need to flip bits in the weight file in the required pages and page offsets. We achieve this in three main steps.

3.4.2.1 Releasing the Flippy Rows

Flipping targeted bits in the model weights requires manipulating the memory mapping of the weight file and placing the target pages to previously found flippy physical addresses. To control the memory mapping, we exploit the *per-CPU page frame cache*. Page frame cache is an optimization implemented in the Linux kernel to utilize hardware caches better in the local CPU by reallocating the recently unmapped page frames in first-in-last-out order [24]. As earlier works showed [33, 110, 240], an attacker process can reliably map the victim page to the recently unmapped pages by exploiting the page frame cache. This unmapping-remapping process is shown by the pseudo-code in Listing 3.1. Although we do not need bit flips in all pages of the weight file, we need the target pages to be mapped to previously

```

1 buffer = mmap(baitPages * PAGESIZE)
2 munmap(flipPageAddr, PAGESIZE)
3 for(i = 0; i < bait_pages; i++)
4     munmap(&buffer[i*PAGESIZE], PAGESIZE)

```

Listing 3.1: Pseudo-code showing how pages can be forced into a specific area in memory

determined flippy page locations. We use a `buffer` with size `baitPages` \times `PAGESIZE` to make sure the parameters we do not target in the weight file are not mapped into the flippy locations. The number of flippy pages and `baitPages` should sum up to the total number of memory pages consumed by the weight file.

We match the target pages in the weight file to the flippy locations and the remaining pages to the non-flippy locations in our buffer. After obtaining a one-to-one mapping between the weight file and our buffer, we start unmapping in the reverse direction to fill the page frame cache.

3.4.2.2 Mapping the Model Weights to Flippy Rows

After releasing the flippy pages and `buffer`, we immediately map the whole weight file from start to end using `mmap` function. The OS automatically maps the weight file to the unmapped locations in the buffer in the right order. An example case is shown in Figure 3.4 for a quantized ResNet20 model. Since all physical addresses match with the released pages of our buffer, there is a one-to-one mapping.

Another way to bring only the target pages of the weight file to the memory is by stating the file offset in the `mmap` function and using `fadvise` with `FADV_RANDOM` flag to prevent the neighboring pages of the file prefetched by the OS, as proposed in [240]. However, in our experiments, we observe that using `fadvise` does not reliably prevent prefetching.

3.4.2.3 Flipping Bits in the Weight File

Finally, the attacker rows are accessed repeatedly to flip bits at the same offsets as found in the offline phase but this time on the weight file. In our experiments, we use n-sided

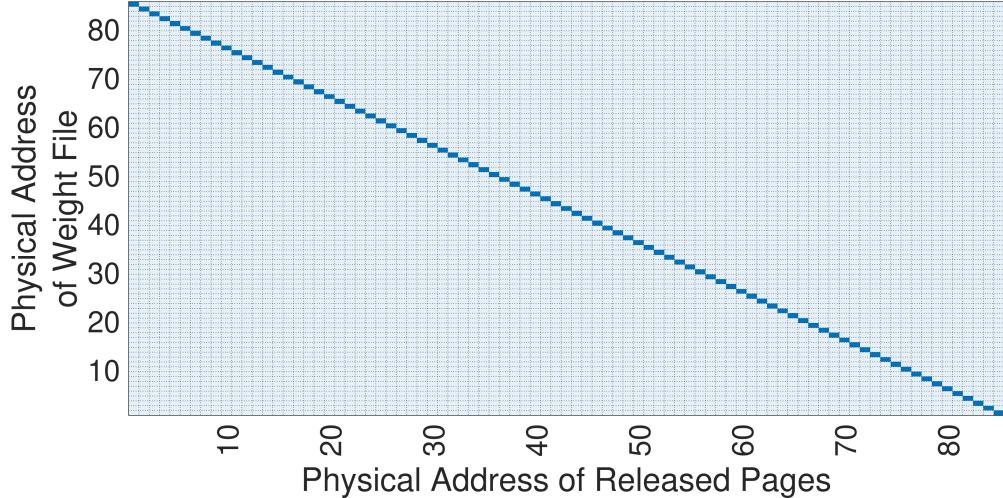


Figure 3.4: Physical Address of released pages vs ResNet20 weight file. First pages of the weight file are mapped to the last released pages of our buffer.

Rowhammer pattern [59] with 7 aggressor rows on DDR4 systems to bypass TRR protection and reproduce the bit flips found in the offline phase. Note that additional bit flips can occur if more than one bit flip is found within a single page. We evaluate the effect of these additional bit flips in Section 3.5.

After completing all the steps in Online Phase, the corrupted weights stay in the memory, and the attacker is able to add the pattern generated in Offline Phase to any image to trigger the backdoor and misclassify the input in a targeted way.

3.4.3 Weight Quantization

The weights are stored as N_q -bit quantized values in the memory as implemented in NVIDIA TensorRT [137], a high-performance DNN optimizer for deployment that utilizes quantized weights [155]. Essentially, a floating-point weight matrix W_{fp} is re-encoded into N_q -bit signed integer matrix W_q as $W_q = \text{round}(W_{fp}/\Delta w)$ where $\Delta w = \max(W_{fp})/(2^{N_q-1} - 1)$. In our experiments, weights are 8-bit quantized and stored in two's complement forms.

3.5 Evaluation

3.5.1 Experimental Setup

To demonstrate the viability of our attack in the real world, we implemented it on an 8-bit quantized ResNet-18 model trained on CIFAR-10 using PyTorch v1.8.1 library. The clean model weights that are trained on CIFAR-10 are taken from [176] for ResNet-18 and from [84] (580 stars on GitHub) for other ResNet models. Moreover, we experimented on larger versions of ResNet models, such as ResNet50, trained on the ImageNet data set. For the models trained on ImageNet, we use pre-trained models of Torchvision library (9.1K stars on GitHub), which has been downloaded 28 million times until now [209]. We run the offline phase of our attack on NVIDIA GeForce GTX 1080Ti GPU and Intel Core i9–7900X CPU. Rowhammer experiments are implemented on DDR3 DRAM of size 2 GB (M378B5773DH0-CH9) and DDR4 DRAM of size 16 GB (CMU64GX4M4C3200C16). The online phase experiments are conducted on a system running Ubuntu 20.04.01 LTS with a 5.15.0–58-generic Linux kernel installed, using a DDR4 DIMM with part number CMU64GX4M4C3200C16. The inference is done on an Intel Core i9-9900K CPU with a Coffee Lake microarchitecture. DRAM row refresh period is kept at 64ms which is the default value in most systems. We use 7-sided Rowhammer to flip bits in the memory. We will provide an explanation for how we decide the number of aggressor rows in Section 3.5.3.

We compare our approach with BadNet [71], and TBT [176] as well as fine-tuning (FT) the last layer. We also include the output of our Constrained Fine Tuning (CFT) without bit reduction in Table 3.2 for comparison. We selected the baseline methods with the aim of creating a backdoor-injected model. We excluded the non-backdoor attacks, such as Deephammer [240], and Terminal Brain Damage [81], in the performance comparison since they only aim to degrade the accuracy of the model. In contrast, we aim to keep the accuracy as high as possible while increasing the Attack Success Rate. For the offline phase results, we keep all the bit flips in the weight parameters assuming they are all viable. In the online

phase results, we keep the bits that are possible to be flipped by Rowhammer and exclude the others. We use 128 images from the unseen test data set for all the experiments in CIFAR-10. TA and ASR metrics are calculated on an unseen test data set of 10K images. In all experiments, we used $\alpha = 0.5$ for Algorithm 1. The trigger masks are initialized as black square on the bottom right corner of the clean images with sizes 10x10 and 73x73 on CIFAR-10 and ImageNet, respectively. ϵ in Equation 3.4 is chosen as 0.001. For the ImageNet experiments, we use 1024 images from the unseen test data set to cover all 1000 classes. TA and ASR metrics are calculated on unseen test data set of 50K.

3.5.2 Evaluation Metrics

Number of Bit Flips (N_{flip}): As in [16, 81, 176, 240], the first metric we use to evaluate our method is N_{flip} , which indicates how many bits are flipped in the new version of the model. The N_{flip} has to be as low as possible because only a limited number of bit locations are vulnerable to the Rowhammer attack in DRAM. As the N_{flip} increases, the probability of finding a right match of vulnerable bit offsets decreases. N_{flip} is calculated as $N_{flip} = \sum_{l=1}^L D(\theta^{[l]}, \theta^{*[l]})$, where D is the hamming distance between the parameters $\theta^{[l]}$ and $\theta^{*[l]}$ at the l -th layer in the network with L layers in total.

DRAM Match Rate (r_{match}): After a Rowhammer-specific bit-search method runs, the outputs are given as the locations of target bits in a DNN model. However, not all of the bit locations are flippable in the DRAM. Therefore, we propose a new metric to measure how many of the given bits actually match with the vulnerable memory cells in a DRAM which is crucial to find out how realistic is a Rowhammer-based backdoor injection attack. r_{match} is calculated as, $r_{match} = \frac{n_{match}}{N_{flip}} \times (1 - \frac{\delta}{S}) \times 100$ where n_{match} is the number of matching bit flips, N_{flip} is the total number of bit flips, S is the number of bits in a page, and δ is the number of accidental bit flips within a page. Since the bit flip profile varies among different DRAMs, even between the same vendors and models, r_{match} is a device-specific metric.

Test Accuracy (TA): In order to evaluate the effect of backdoor injection to the main

task performance we use Test Accuracy as one of the metrics. Test Accuracy is defined as the ratio of correct classifications on the test data set with no backdoor trigger added. Ideally, we expect the backdoor injection methods to cause minimal to no degradation in the Test Accuracy in the target DNN models.

Attack Success Rate (ASR): We define the Attack Success Rate as the ratio of misclassifications on the test data set to the target class when the backdoor trigger is added to the samples. Attack Success Rate indicates how successful a backdoor attack is on an unseen data set.

3.5.3 Rowhammer Attack on Deployed Model - Online

We experiment the online phase of the attack on DDR3 and DDR4 DRAM chips. We empirically observe that when there are multiple bits required to be flipped on the same 4KB page in a particular direction ($\{0 \rightarrow 1\}$ or $\{1 \rightarrow 0\}$), there is no matching target page in the 128 MB Rowhammer profile. This observation shows that multiple bit flips at desired page offsets and bit-flip direction is an unrealistic assumption. On the other hand, we observe that there is always a matching page in the profiled memory buffer with a bit flip in the desired location and flip direction if there is at most one bit flip in the memory page. This observation is consistent with our probability analysis in Section 3.4.1.2. Apart from the targeted bit flips, we observed that some DDR4 DRAMs with large average bit flips in a page give accidental bit flips in addition to the target offsets which reduces the r_{match} .

Effect of Number of Attacker Rows on Bit flips The idea of a multi-sided Rowhammer attack is that instead of a single row above and below the victim row being read, another victim is created above the attacker row, and another attacker above the new victim a variable number of times. Figure 3.5 shows how the number of attacker rows changes the bit flip rate.

Figure 3.6 shows that by reducing the number of aggressors in n-sided Rowhammer from

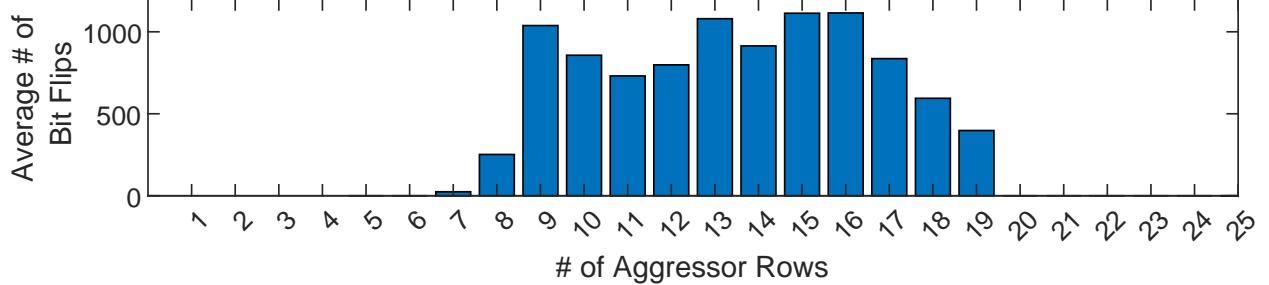


Figure 3.5: Average number of bit flips on an 8MB buffer vs the number of sides in an n-sided Rowhammer attack.

15 to 7, we can reduce the number of additional flips to 4 bits per target page. Therefore, we use 7-sided Rowhammer in the later experiments. Random bit flips outside the target location have a very limited effect on both TA and ASR since the target model weights are quantized [81].

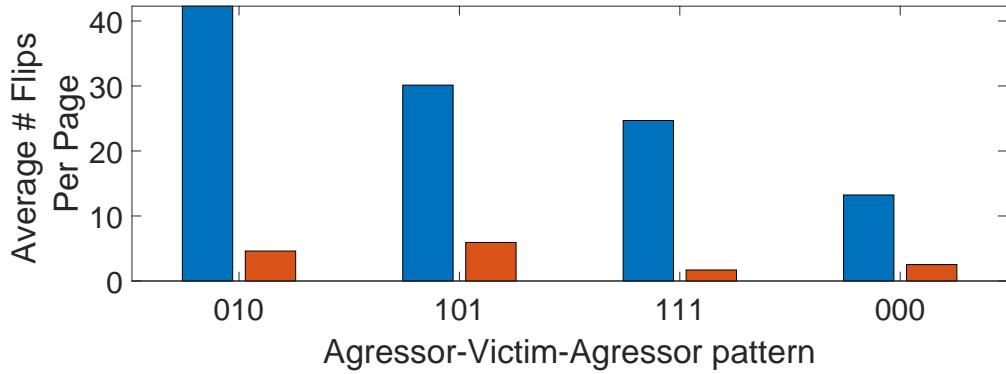


Figure 3.6: Average number of bit flips per page for 15-sided (blue) and 7-sided (red) Rowhammer attack patterns.

As shown in Table 3.2, we get 99.9% r_{match} for every DNN model we attack with CFT+BR since all of the required bit flips we need are in separate pages. Whereas BadNet, FT, TBT, and CFT, have very low numbers achieving as low as 1 bit flip since they require multiple-bit flips with specific locations and flip directions in the same memory page.

Table 3.2: Comparison of our methods CFT, CFT+BR with the baseline methods BadNet, FT, and TBT on CIFAR10 [109] with ResNet-20/32/18, and ImageNet [184] with ResNet-34/50. Our proposed CFT+BR results are written in bold. Note that the percentage of the backdoor parameter bits ($\Delta\theta$) that are actually flippable, r_{match} , must be near 100% for a viable backdoor injection attack using Rowhammer.

Dataset	Net	Method	Offline Phase			Online Phase		
			N_{flip}	TA(%)	ASR (%)	N_{flip}	TA (%)	ASR (%)
CIFAR10	ResNet20	BadNet	172,891	86.96	99.98	33	91.76	2.63
	Acc: 91.78%	FT	2,238	84.36	97.10	1	91.72	2.90
	#Bits: 2.2M	TBT	44	86.61	95.43	1	91.72	4.71
	#Pages: 69	CFT	22	90.09	99.55	5	91.79	14.40
	ResNet32	CFT+BR	10	91.24	94.62	10	89.04	92.67
	Acc: 92.62%	BadNet	246,004	88.60	99.99	53	92.61	7.32
	#Bits: 3.7M	FT	2318	81.87	90.59	1	92.65	8.57
	#Pages: 116	TBT	210	81.90	89.66	1	92.66	8.42
	ResNet18	CFT	39	90.25	98.75	10	92.41	20.22
	Acc: 93.10%	CFT+BR	95	91.77	91.46	95	89.56	89.58
	#Bits: 88M	BadNet	1,493,301	87.61	99.88	416	93.06	12.45
	#Pages: 2750	FT	8,667	88.80	95.34	1	92.20	34.16
ImageNet	ResNet34	TBT	95	82.87	88.82	1	92.60	48.12
	Acc: 73.31%	CFT	42	92.39	99.90	11	91.52	0.36
	#Bits: 172M	CFT+BR	99	92.95	95.26	99	90.71	93.30
	#Pages: 5375	BadNet	441,047	70.81	99.73	100	70.39	0.009
	ResNet50	FT	54,726	68.30	99.14	11	70.95	0.18
	Acc: 76.13%	TBT	553	72.69	99.86	1	70.97	0.05
	#Bits: 184M	CFT	1509	70.25	99.76	388	69.93	0.10
	#Pages: 5750	CFT+BR	1463	70.28	72.92	1463	68.59	71.42
	ResNet50	BadNet	359,516	73.98	99.11	129	66.43	0.05
	Acc: 76.13%	FT	93,778	68.43	96.52	12	73.77	0.09
	#Bits: 184M	TBT	543	75.60	99.98	1	73.78	0.10
	#Pages: 5750	CFT	1562	70.58	99.99	391	66.71	4.92
	ResNet50	CFT+BR	1475	70.64	98.22	1475	68.94	96.20
	Acc: 76.13%	BadNet	359,516	73.98	99.11	129	66.43	0.05
	#Bits: 184M	FT	93,778	68.43	96.52	12	73.77	0.09
	#Pages: 5750	TBT	543	75.60	99.98	1	73.78	0.10

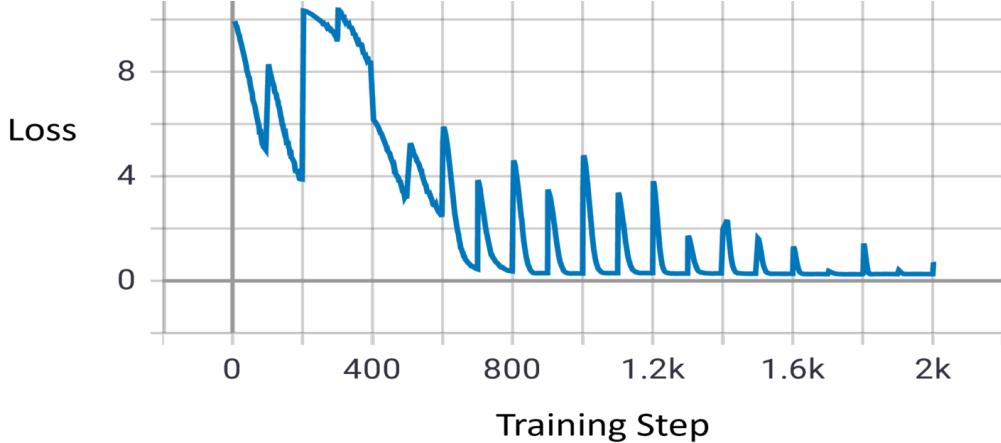


Figure 3.7: Total loss graph at every training iteration during the backdoor injection to the ResNet18

3.5.4 CIFAR-10 Experiments

We experiment with our proposed method on ResNet18, ResNet20, and ResNet32 trained on CIFAR-10 along with the baseline methods, such as BadNet, FT, and TBT. We also compare our partial method, CFT, with our complete method (CFT+BR) which includes the *Bit Reduction*. During the iterations of CFT+BR, we observed that the total loss spikes after each *Bit Reduction* and quickly decreases again and eventually converges to a solution $\theta + \Delta\theta$ as described in Equation 3.6. Figure 3.7 shows the loss progress after each epoch with one batch of data while optimizing a constrained weight perturbation $\Delta\theta$ to a ResNet18 model on the CIFAR-10 data set. After every 100 iterations, we apply *Bit Reduction*, which causes spikes in the loss curve. We compare our method with baselines for both phases since our attack scenario includes offline and online phases. Recall that in the offline phase, the optimization takes place to find the vulnerable bit locations and generate a trigger pattern. First, we evaluate the modified models with the corresponding trigger patterns. Then, for each modified part of the weight parameters, we look for a matching target page location on the profiled memory, which constitutes the online phase. If multiple bits need to be flipped in the memory, we choose the one with the largest gradient value so that we get the maximum possible performance from the baselines. Finally, DRAM Match Rate r_{match} is calculated as

explained in Section 3.5.2. The experiment results are summarized in Table 3.2.

BadNet and FT have no control over the N_{flip} since they do not introduce any constraints during the optimization. Therefore, in the offline phase, BadNet requires up to one and a half million bit flips to inject a backdoor successfully. Although FT modifies only the last layer while keeping the other layers constant, meaning fewer bit flips than BadNet, we observe that up to 8,667 bits have to be flipped. TBT has control on the number of modified parameters which enables partial control on the N_{flip} since the number of modified parameters limits the maximum value N_{flip} can get. Therefore, we select the results that reproduce their claimed performance in the original work [176] without modifying too many weight parameters and increasing the N_{flip} too much, and thus, decreasing r_{match} further. In the offline phase, TBT finds a much smaller number of bits compared to BadNet and FT due to the limit on the modified parameters. Our experiments show that the CFT+BR method successfully injects a backdoor into ResNet20 model with **91.24%** TA and **94.62%** ASR by flipping only **10 bits** out of 2.2 million bits in the DRAM. In ResNet32 and ResNet18, CFT+BR achieves 91.46% and 95.26% ASR, respectively, with a maximum of 1.66% degradation in the TA. We observe that N_{flip} values in BadNet and FT depend heavily on mode size. As the total number of bits increases, they require more bit-flips to achieve similar performance. On the other hand, we do not observe a significant dependence on the model size in TBT, CFT, and CFT+BR methods in terms of N_{flip} , TA, and ASR. In BadNet, FT, and TBT, the bit flips are concentrated within the same pages. Especially FT and TBT targets on the last layer of the DNN models. Since the last layer of the Resnet20, ResNet32, and ResNet18 models occupy only one memory page in DRAM, the bit-flip locations found in the offline phase of FT and TBT reside within a single page. For instance, 210 bit-flips found by TBT on ResNet32 are all on the same page. However, as we mention in Section 3.4.1.2, only the pages with one targeted bit location can be found in DRAM in practice. Therefore, we choose the bit flip with the largest gradient in a memory page and keep it modified and return the other parameters to their original values. Finally, we evaluate their performance on the

test data set. In the ResNet20 and ResNet32 models, we observe that the ASR of BadNet, FT, and TBT drops down below 10% while the Test Accuracy values increase back to their original values. We claim that the significant decrease in ASR values can be explained by the diffusion effect of optimizing the parameters in an unconstrained way. When the attack is implemented on DRAM using Rowhammer, r_{match} values of BadNet, FT, and TBT are lower than 3% for every DNN model. In CFT, r_{match} is relatively higher than the other baseline methods since it modifies only one parameter in a page. However, it does not put a constraint on the number of bit flips within a byte during the optimization. Therefore, the attack performance degrades drastically in practice. In all experiments, CFT+BR has 99.9% r_{match} since it already considers the bit locations that can be flipped during the attack. Since the bit flips are sparse across different memory pages in CFT+BR, **100%** of the bit flips can actually be flipped. A small number of bits may be flipped in random locations, but it does not affect the performance of the attack significantly. We show that lower r_{match} values lead to low ASR in backdoor injection attacks using Rowhammer.

3.5.5 ImageNet Experiments

We also compare our method with the baseline methods on models trained on the ImageNet ILSVRC2012 Development Kit [184] data set, which consists of 1000 classes of visual objects. We used pre-trained ResNet34 and ResNet50 from the model zoo [209] as the target models. ResNet34 and ResNet50 include 172 million and 184 million bits, respectively. Note that both the model and data set sizes are significantly larger compared to our CIFAR-10 experiments. As the TA and ASR, we use top-1 accuracy results. The results are summarized in Table 3.2. The same comparison methods we apply in CIFAR-10 are valid in ImageNet experiments as well.

In the offline phase of the attacks, we observe that each method shows a different response to the increase in the model and data set sizes. For instance, BadNet and FT require more than 350K and 50K, respectively. Compared to CIFAR-10 models, BadNet is not

affected significantly. However, N_{flip} for FT becomes 17 times larger on average on the ImageNet models. TBT locates around 550 N_{flip} on the ResNet34 and ResNet50 models in the offline phase, which is 5 times larger on average than the CIFAR-10 experiments. CFT and CFT+BR locate around 1500 N_{flip} on the ResNet34 and ResNet50 models in the offline phase, meaning 45 times and 22 times larger for CFT and CFT+BR, respectively.

In the online phase, we observe that none of the baseline methods has a significant attack performance. For instance, in the BadNet method, although the model sizes increase 5.5 times, the number of modified pages increases only 1.5 times on average. Similarly, TBT modifies only one page in the last layer of the ResNet34 and ResNet50 models, even though the last layers of the models have more than 10 pages. This clearly shows that as the model size increases, the density of bit flips required by the baseline models increases, meaning the attack tends to focus on certain regions instead of uniformly distributing the bit flips. The high density of the bit flips leads to r_{match} rates as low as 0.02%. Although FT modifies most of the pages in the last layer, the fact that the bit locations are not optimized at the beginning causes vanishing ASR. Overall, we observe that the claimed ASRs can be achieved only when r_{match} is large enough. Although CFT achieves much larger r_{match} values than the other baseline methods, lacking *Bit Reduction* makes the attack focus on multiple bit flips within 8-bit parameters, which, in return, causes lower than 5% ASR on the models trained with ImageNet data set. In contrast, CFT+BR can inject the backdoor to ResNet models with up to 96.2% ASR and a maximum of 7.2% degradation in the TA, which makes it the best-performing backdoor injection attack compared to the baseline methods. These results show that our approach generalizes well to larger data sets and models. Note that although N_{flip} increases as the model gets larger in CFT+BR, it is still possible to flip these bits with 99.99% r_{match} due to the sparse distribution.

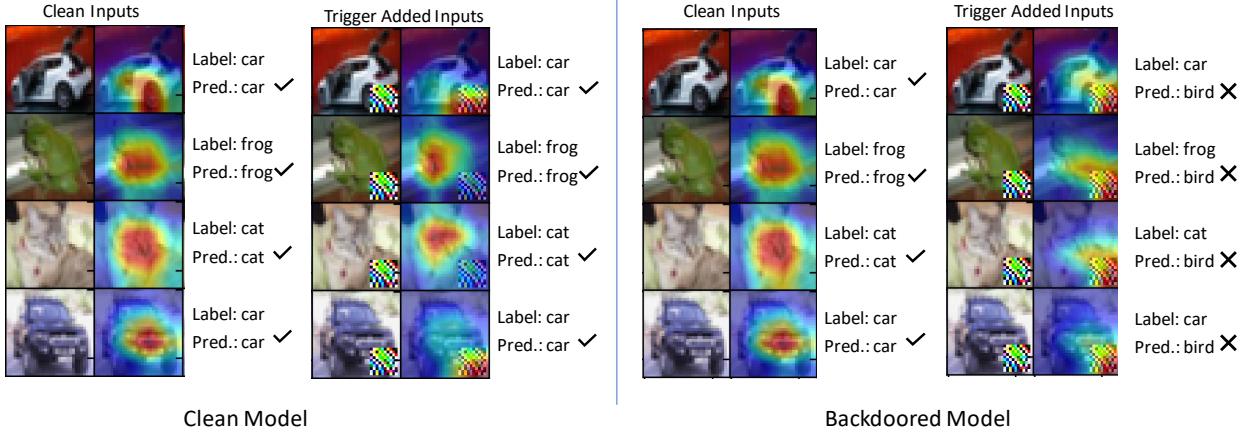


Figure 3.8: The change in GradCAM [190] heatmaps that belong to ResNet18 before the attack (left) and after the attack (right). The focus of the model shifts through the trigger pattern if it is backdoored.

3.5.6 Generalization to Other DNN Architectures

We experiment on other DNN architectures, such as VGG11, VGG16, to show that our attack generalizes. We show that CFT+BR can successfully locate vulnerable bits and achieves over 90% Attack Success Rate in VGG architectures. The results are summarized in Table 3.3.

Table 3.3: CFT+BR experiment results on VGG architectures

Model	Base Acc	TA [%]	ASR [%]	N_{flip}
VGG11	92.35	92.70	100	30
VGG16	92.68	92.57	90.85	100

3.6 Potential Countermeasures

We analyze some of the prominent countermeasures proposed for mitigating bit-flip attacks against DNN models.

3.6.1 Prevention-Based Countermeasures

Binarization-Aware Training [79]¹ is a method that uses *Binarized Neural Networks* (*BNNs*) [83, 178] to increase the resistance of DNNs against the bit flip attacks. This method significantly reduces the network size. For instance, a binarized ResNet-32 model occupies only 65 pages in the memory. Although 65 bit flips are not enough to inject a backdoor using Rowhammer, N_{flip} cannot be larger than the number of pages occupied by the model. Therefore, our experiments show that using BNNs is an effective defense against our attack since it aggressively decreases the size of the network and, consequently, the maximum N_{flip} . However, reducing the model size causes accuracy degradation as a performance overhead. Note that BNNs may still be vulnerable to other fault attacks which do not require the same physical constraints, such as sparse faulty bit locations.

Piecewise Weight Clustering (PWC) [79] is a relaxation of BNNs. With PWC, an additional penalty term is introduced to the inference loss function, which forces model weight distribution to form two clusters. We experiment with our attack against a ResNet32 model trained with PWC penalty term in the loss function. We observe a strengthened trade-off between the TA and ASR during the optimization. For instance, the ASR drops down to 43.42% when TA is 89.66% with 112 N_{flips} . On the other hand, our attack achieves 98.49% ASR while degrading the TA down to 9.9% with the same N_{flips} . The results show that training the model with PWC does not protect against accuracy degradation and even targeted misclassification attacks. However, it makes it harder to inject stealthy backdoors.

3.6.2 Detection-Based Countermeasures

Possible defense techniques focusing on detecting the attacks on the model weights [40, 115, 118, 124] come with an overhead because they need to be deployed together with the model into the machine learning product.

¹Binarization-Aware Training and Piecewise Weight Clustering implementations are taken from <https://github.com/elliothe/BFA>.

DeepDyve [118] is a dynamic verification method that uses a checker model along with the original model for mitigating the transient faults in the inference. It assumes both models predict the same results for the same inputs most of the time. When the results of the two models are the same, the result is accepted immediately. However, if the results are different, the inference is repeated, and the second result from the original model is accepted. DeepDyve assumes the fault in the model is transient and does not appear in consecutive queries. However, the bit flips introduced by Rowhammer stay in the memory until being reloaded from the disk. Since the transient assumption does not hold, even if a checker model raises an alarm and repeats the inference, the new inference is made by the backdoor-injected model and will not be detected.

Weight Encoding [124] proposes additional matrix multiplication and weight extraction. Thus, this method can detect only the topmost sensitive layers in the network to keep the overhead low. However, our attack can target all layers to inject a backdoor. Therefore, the spatial locality assumption does not hold with our attack. Using the overhead numbers in [124] for ResNet-34, we estimate the time and storage overhead against our attack. Since the time complexity of weight encoding $d_j = r(y_j), y_j = \phi(\sum_{i=0}^{N-1} B_i \cdot K_{ij})$ is $O(N^2)$, where B is Z^N , and K is $R^{N \times M}$, the estimated execution time overhead of the method is 834.27 seconds. Since the storage complexity of the Weight Encoding is linear, the storage cost for ResNet34 is estimated as $(0.141/8192) \times 21779648 = 374.86MB$, which is 446% storage overhead, showing that the proposed method is not scalable.

RADAR [115] is a checksum-based detection method during inference. It divides the weights into groups and gets the checksum of the most significant bits of parameters in each group. The original checksum values of the parameters are stored along with the model and are validated with the original signatures at every inference time. The optimization constraints can be further increased to avoid flipping the MSB of the weight parameters in our attack, which can bypass the detection. Assuming linear time complexity, time overhead goes up to 40.11% for full-size bit protection in ResNet20.

SentiNet [40] filters the adversarial inputs using GradCAM heatmaps [190]. We use the GradCAM implementation from [64] to analyze the output of four sample images that are labeled as *car*, *frog*, *cat* and *car* respectively (See Figure 3.8.). Before the attack, the model correctly classifies all images with or without the trigger pattern. If the trigger pattern does not overlap with the major features in the image, e.g. *frog* and *cat*, the main focus of the model stays on the object. However, if the trigger pattern overlaps with the main features, e.g. the wheel of the *car*, the focus is shifted towards the trigger pattern. After the attack, regardless of the trigger and object overlap, the focus of the model shifts towards the trigger pattern, and the model misclassifies all images to the target class, *bird*. Therefore, although a GradCAM-based approach can possibly filter the adversarial inputs, it will also produce false positives even if the model is clean and works correctly.

3.6.3 Recovery-based Countermeasures

Weight Reconstruction: Li et al. [116] propose *Weight Reconstruction*² to recover the clean network after a bit flip attack occurs. *Weight Reconstruction* aims to recover from an accuracy degradation caused by the attack. After a bit flip occurs in a weight parameter, the effect of the change is distributed onto other parameters to reduce the overall effect on the model performance. We experiment with our CFT+BR attack against a ResNet32 defended by Weight Reconstruction to evaluate the effectiveness of the proposed defense method. We applied our attack in two different scenarios. In the first scenario, the attacker is not aware that the model is defended by Weight Reconstruction and applies the offline phase of the attack as described in Section 3.4.1.3. As a baseline, our attack achieves 91.46% ASR and 97.77% TA by flipping 95 bits in the memory. After applying Weight Reconstruction, we observed that ASR and TA become 32.89% and 91.02, respectively. In the second scenario, the attacker is aware that the model is defended by Weight Reconstruction and applies the offline phase of the attack against a model with Weight Reconstruction. However, if the attacker

²Weight Reconstruction implementation is taken from https://github.com/zlijingtao/DAC20_reconstruction.

is aware of the defense and applies CFT+BR on a defended model, our attack successfully bypasses Weight Reconstruction by achieving 94.04% ASR and 89.51% TA. Therefore, the Weight Reconstruction approach does not protect the models when the attacker knows the applied defense.

3.7 Related Works

Rowhammer Attacks on DNNs We compare our work with Terminal Brain Damage [81] and Deephammer [240] in terms of the following factors:

Attacker’s Objectives: The main difference between our work and previous works is the goal of the attack. In both [81] and [240], the attacker’s objective is to degrade the inference accuracy of the model on legitimate inputs and cause a denial of service. In contrast, our attack objective in this work is to keep the inference accuracy for legitimate inputs the same and misclassify all trigger-added inputs to a target class in stealth by using a unified objective function given in Equation 3.3.

Assumptions: All [81], [240], and our work assume the attack takes place in a cloud environment where the model is loaded into system’s shared memory and stays unchanged. Unlike [240], we do not assume the availability of huge page configuration to bypass virtual to physical translation.

Attacker Capabilities: Same as our attack, [240] and [81] assume the attacker knows the model architecture and parameters. [81] also considers black-box setting with random bit flips. Since our attack objective is more sophisticated, our attack is not applicable in a black-box setting.

Attack Time: [240] configures the hammering time for each row as 190ms. Since [81] only simulates the attack, they assume it is 200ms in the calculations. In our setup, it takes 800ms to hammer one row using a 15-sided pattern during the profiling phase and 400ms using a 7-sided pattern during the online phase. Note that previous works consider

only double-sided Rowhammer, which takes less time but is not effective on DDR4 chips with TRR mitigation. Total online attack time varies between different models and can be estimated by multiplying the hammering time by N_{flip} .

Stealthiness/Detectability: Due to the difference in the attack objectives, the stealth of the attacks is also different. For instance, Test Accuracy after the attack on VGG16 is given as around 10% in both [81] and DeepHammer. However, we can preserve the Test Accuracy at over 92% after our attack while being able to misclassify over 90% of all instances with an attacker-generated trigger pattern. Since we can preserve the Test Accuracy close to the base accuracy of the models, our attack is stealthy.

Comparison of Accuracy Degradation: Although the goal of backdoor injection is not accuracy degradation, the resulting degradation on trigger-added inputs is comparable to [81] and [240]. In VGG16 trained on CIFAR10, when we add trigger pattern to all images, we see the accuracy of the model to be 18% (an 80% relative accuracy degradation from baseline). Alternatively, [240] and [81] claim relative accuracy degradations of VGG16 to be 88% and 90%, respectively (after the attack the models only produce a correct output 10% of the time).

Accuracy Degradation Attacks Bit-Flip Attack [175] degrades the accuracy of DNN models to random guess using a chain of bit flips. Targeted Bit-Flip Attack [177] is shown to be capable of misclassifying the samples from single or multiple classes to a target class on quantized DNN models. Although these works show that DNN model performance can be damaged permanently by flipping a limited number of bits in the weight parameters, these attacks do not make use of an attacker-controlled backdoor trigger. Therefore, they have very limited control over stealthiness. A binary integer programming-based approach was proposed by Bai et al. [16] to find the minimum number of bit flips required to make the model misclassify a single image sample into a targeted class.

ML Backdoor Attacks Garg et al. [60] observed that adversarial perturbations on the weight space of the trained models could potentially inject Backdoor, but it requires either social engineering or full privileged access to replace the target model with the backdoored model. [176] and [37] showed that backdoor attacks could be implemented by changing only a small number of weight parameters. However, both of the works assume any bit location in the memory can be flipped, which is not practical. Therefore, the practicality of software-based backdoor injection attacks during the inference phase is still an open question due to the practical constraints that have been overlooked in previous works.

3.8 Discussion

Alternative Methods for Identifying Target Bits: The CFT+BR algorithm presented in this work leverages domain-specific knowledge about Rowhammer and DNN behavior under perturbations. While traditional optimization methods such as Bayesian Optimization (BO) could theoretically be adapted for this purpose, they present significant challenges. Specifically, BO struggles with high-dimensional search spaces, such as the millions of bits in memory, and requires surrogate models that scale poorly with dimensionality. Moreover, BO is not inherently designed for discrete and heavily constrained problems, which are critical in this context. Although BO’s sample efficiency and ability to handle expensive evaluations may be advantageous in smaller sub-problems, it would require extensive customization to address the constraints of this attack scenario.

Effect of Huge Pages: We assume huge pages are not available since they give an advantage for finding contiguous memory in physical address space. Even though the target model uses huge pages, the memory controller would still fragment the huge page into 8 KB rows in DRAM due to the fixed row size. Also, each chunk is mapped into different banks in order to increase parallel access. For example, if there are 64 banks in the system, a 2 MB huge page would be fragmented into 64 chunks and 4 neighbor rows in the DRAM. Although

this may hurt the n-sided Rowhammer pattern, it would still be possible to sandwich each chunk and do Rowhammer. Note that, in memory systems with multiple DIMMs, and ranks, the number of banks also increases, which would decrease the size of the chunks down to a single row. In that case, a regular double-sided or n-sided Rowhammer attack would still work. Since an attacker can choose to profile 4 KB pages in DRAM, finding 512-bit flips in 2 MB would still be practical.

Application on Other Security Critical Tasks: The proposed attack method is a generic approach and agnostic to the downstream tasks. Therefore, it would work on models used in other safety-critical tasks, such as voice recognition applications.

3.9 Dynamic Analysis Approach on the Detection of Fault Targets

While numerous studies have extended Rowhammer’s applicability to areas like cloud environments and network-based attacks, recent advancements include targeting sensitive CPU stack variables to execute malicious code and bypass security measures. In this section, we investigate the automatic detection of *LeapFrog* gadgets that enable Rowhammer to break security assumptions in cryptographic libraries and machine learning models. *LeapFrog* gadgets allow the manipulation of the program counter (PC) stored in the stack to subvert control flow, bypassing critical security features like authentication and encryption.

Based on how the *LeapFrog* gadgets occur in the binary, we developed a custom tool we call MFS (Multidimensional Fault Simulator) that relies on dynamic binary instrumentation and analysis. Since the attack happens on program counters and registers, which are invisible to high-level code, such as C/C++, it is not possible to do a static analysis of the source code. We put together a set of rules that enables us to collect, filter, and pinpoint the potential *LeapFrog* gadgets. The overall design is shown in Figure 3.9.

- ❶ First, MFS collects the instruction traces, specifically, the address of instructions

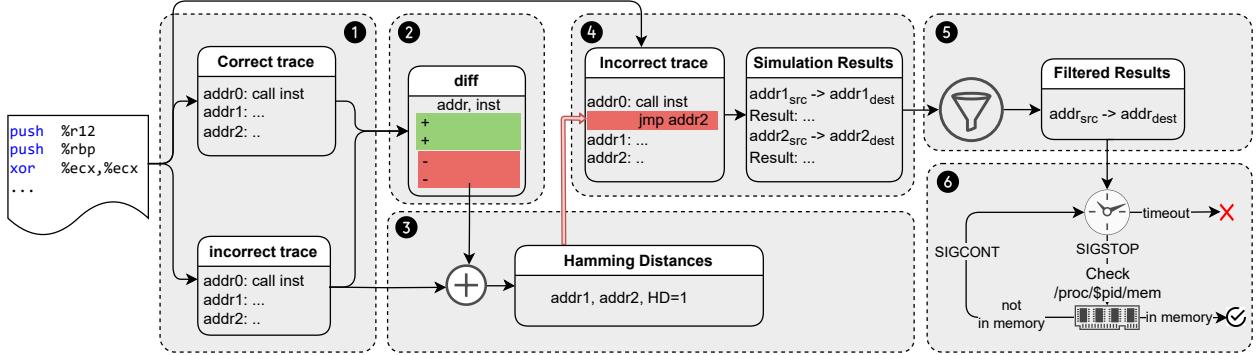


Figure 3.9: LeapFrog gadget detection using MFS framework

executed, for different inputs. For the purpose of detecting the gadgets that cause security exploits, MFS chooses critical input pairs that cause differences in the program’s control flow. Such inputs can be correct/incorrect private key pairs or passphrases for authentication programs. Together with the instruction addresses, we collect the execution time of each function executed. Since the return addresses of the functions with larger execution times will stay in the memory for a longer duration, they are potentially more viable targets.

② MFS then computes the difference between two instruction traces to find the instruction addresses that are executed with correct input(s) but not executed with incorrect input(s). Note that this is an optional step to reduce the complexity of the following steps, and it comes with a cost of false negatives. Moreover, depending on the program and type of exploit, it may not always be possible to get multiple different traces; see §3.9.2.2. Alternatively, the whole instruction trace can be considered instead of only the difference.

③ MFS then looks for address pairs that hold the following conditions:

$$d_H(\text{addr}_{\text{exec}}^i, \text{addr}_{\text{return}}^j) = 1 \quad (3.7)$$

where $\text{addr}_{\text{exec}}^i$ is the address of the i^{th} instruction that is executed, $\text{addr}_{\text{return}}^j$ is the return addresses of the j^{th} call instruction, and d_H is the Hamming distance between two addresses. i and j are bounded by the number of all instructions executed (n) and the number of call instructions executed (m), respectively. Although this operation has $O(m^n)$ complexity, it

can be implemented with bitwise xor and can be parallelized using multiple processor cores. The condition given in Equation equation 3.7 is determined by the Rowhammer fault model. Since multiple-bit flips on a precise target are much rarer and harder to control, MFS assumes we can only flip a single bit. Yet, the method is generic enough to cover other potential fault models, such as optical fault injection, where multiple-bit flips are more likely [25]. This step generates a list of pairs of addresses in the following format: $\{<addr_{src}^k, addr_{dest}^k>\}$ where $addr_{src}^k$ is the k^{th} instruction address that MFS targets in the binary's execution with the input that we want to affect the control flow of, such as an incorrect private key, and $addr_{dest}^k$ is the corrupted instruction address after fault injection.

④ For each address pair we get from the list generated in the previous step, MFS starts a simulation session. MFS executes the binary again with the incorrect input and simulate a bit flip on the instruction address $addr_{src}$ to make it $addr_{dest}$. It is possible that certain instructions are executed multiple times in a single execution. To correctly cover that case in our fault model, we keep a counter variable for a specific instruction that increments every time the binary executes the same instruction. In a single execution of the original binary, if an instruction is executed N times, we attempt the fault simulation $N + 1$ times, until we no longer see the same instruction in the trace.

⑤ After the bit flip simulation, MFS continues the execution of the binary without further faults and observe the new behavior. The analysis of the new behavior is not a trivial task. There are several options where we can observe changes compared to the original execution. For instance, we can observe changes in the total number of executed instructions, the number of instructions that match with the correct input execution trace, the return code of the program, outputs to standard streams, ports that are accessed, functions calls, authentication result, etc. The choice of observable depends on the program under test. In this work, MFS uses the return codes, standard outputs/errors, and authentications on different case studies.

3.9.1 Tool Implementation

We used Intel’s dynamic binary instrumentation framework, `Pin`, which allow for process analysis without altering its core behavior [128] to implement ① and ④ of MFS. Using `Pin` also makes it possible to find LeapFrog gadgets in binaries that do not have a source code since it does not require recompiling. In the context of MFS, `Pin`’s capabilities are harnessed to monitor the execution trace of a binary. This integration allows for a thorough analysis of potential LeapFrog gadgets by observing how changes in PC values influence program behavior. For each executed instruction, our tool outputs the virtual address of the instruction and disassembly of the machine code. If the instruction is a call instruction it also outputs the return address of the call, which is usually the PC value that is pushed onto the stack before executing the called routine. For every write to `STDOUT` and `STDERR`, the tool forwards a copy of the buffer to a text file for further analysis. To avoid the effect of overhead caused by instruction-based instrumentation, function timings are collected in a separate session on every function entry and exit.

② is a simple comparison operation on the correct and incorrect execution traces implemented with `diff` command line tool in Linux.

③ is implemented in Python. MFS parses the instruction traces and computes the Hamming distance between the return addresses and instruction addresses of all executed instructions in the correct trace or the list of addresses we get from ②. The Hamming distances are calculated using the native `bit_count` function in Python followed by `bitwise_xor` in `numpy` library. The operation is parallelized on multiple cores to speed up the analysis.

The bit flip simulation part of MFS (④) is done using `Pin` which takes the address pairs and simulates every fault independently. The faults on PC values are implemented as direct jumps to the corrupted addresses by adding `jmp addrdest` after function returns. Since we add a direct jump to the target address by injecting a line of assembly with the the `Pin` tool, it is functionally equivalent to corrupting the PC value in memory.

⑤ filters the simulation results depending on the program and targeted exploit type. For

different types of exploits, we can filter by return code or value in STDOUT.

3.9.2 Experiments

Experiment Setup The experiments are conducted on a system with Ubuntu 22.04.2 LTS with 6.2.0-37-generic Linux kernel installed. The system uses an Intel Core i9-9900K CPU with a Coffee Lake microarchitecture. We used a dynamic clock frequency rather than a static clock frequency to improve the practicality of the attack. End-to-end attack experiments are done on a single DIMM Corsair DDR4 DRAM chip with part number CMU64GX4M4C3200C16 and 16GB capacity. DRAM row refresh period is kept at 64ms, which is the default value in most systems. In all the experiments, we used 100s simulation timeout, since the fault simulations rarely cause infinite loops. We empirically observe that using the Python signals library, the target process could complete 34M cycles before the attacker can stop it, with a standard deviation of 2.7M cycles. Alternatively using a bash script, the victim process can only complete 18M cycles before it is stopped, with a standard deviation of 0.3M cycles. There is an order of magnitude difference in precision stopping a process with bash vs with Python.

3.9.2.1 ML Misclassification

In this section, we investigate the potential implications of instruction skipping in the machine learning domain, specifically for decision tree algorithms. A decision tree is an ML model used to make predictions based on a series of binary choices, effectively splitting data into increasingly specific groups. It starts with a single node, which branches into possible outcomes based on the features of the data. Each branch represents a decision pathway, and each node in the pathway represents a test on a specific attribute. This process continues until a leaf node is reached, which provides the predicted outcome. They are widely used in various applications, from financial forecasting [123] to medical diagnosis [192] due to their interpretability, and efficiency for a variety of tasks such as classification, and feature

importance ranking. We choose a decision tree for proof of concept yet instruction skipping attacks can be effective in every kind of model implementation.

Classification algorithms may be vulnerable to the LeapFrog attack under the threat model that an attacker is co-located on the server with the victim process running the model, and the attacker would like to force a particular output. If the attacker faults the victim process program counter and forces a jump in the code, the result may be a misclassification or a forced classification of a particular output. This attack is different from other Rowhammer attacks on machine learning models [207] because for this attack we do not need to know the model weights before hand, and we consider this a gray box model.

In this experiment, we use a public implementation [157] as our target. We simulate program counter flips and observe the effects on the model output. We follow a similar procedure to previous examples, where we experiment with a hammering distance of 1, 2, and 3 and determine the number of successful LeapFrog gadgets with each of these distances. In Table 3.4, we can see various number of LeapFrog candidate gadgets that might result in a misclassification. After simulating these gadgets, we found 23 of the 1363 potential gadgets within 1 hammer distance would result in a misclassification.

Table 3.4: Number of gadget candidates found in decision tree algorithm with different Hamming distances.

Target	Size	#Inst. _{exec}	d_{HD}	# Candidates	
				❷ on	❷ off
Decision Tree	99KB	38417	1	N/A	1363
			2	N/A	8667
			3	N/A	32326

3.9.2.2 Crypto Libraries

OpenSSL Encryption Bypass We analyze `openssl` command line tool that uses OpenSSL v1.1.1w for block cipher and stream cipher implementations. For each cipher, we give a simple plaintext that contains the `helloworld` string and run encryption without salt with a

Table 3.5: Number of gadget candidates found by MFS on OpenSSL for fault models with different hamming distances. We ran OpenSSL with `aria-128-cbc` cipher.

Target	Size	#Inst.exec	d_{HD}	# Candidates	
				② on	② off
OpenSSL	818KB	49431	1	N/A	2700
			2	N/A	20208
			3	N/A	70475

simple passphrase. Our aim is to find LeapFrog gadgets in the binary that can be exploited for bypassing encryption steps in the ciphers, revealing the plaintext.

First, we scan the binary using MFS as described in §3.9. Since we do not aim for any authentication bypass in this scenario, and the execution traces are deterministic for fixed inputs, step ② is not applicable. Instead, in step ③, we compare the return addresses in a single trace against all the instruction addresses in the same trace to look for targets with $d_{HD} = 1$. The results for `aria-128-cbc` summarized in Table 3.5.

We scanned the binary with 135 different ciphers available in OpenSSL. Most of the time the binary was not affected by the simulated bit flip and correctly produced the ciphertext.

Fig. 3.12 illustrates one of the LeapFrog gadgets found in the `openssl` command line tool. When we corrupt a single bit in `0x55555559c4c5`, the return address of `opt_cipher` function, to make it `0x55555559c0d5`, the function returns to the corrupted return address, skipping three instructions in between. Similarly, another single-bit corruption to (`0x55555559c0c5`) causes the function to return to an earlier point in the program. We verified that both of these bit flips cause the binary to skip the whole encryption and instead output the plaintext. Similarly, MFS detected LeapFrog gadgets that are used in 36 ciphers including block ciphers and stream ciphers. The ciphers with LeapFrog gadgets that revealed full or partial plaintext are listed in Table 3.6. Fig. 3.10 and 3.11 summarize the simulation results for `aes-256-ctr` and `aria-256-ctr` respectively.

Even with ASLR enabled, these gadgets are reproducible because ASLR does not randomize the last 12 bits of the code space (the page offset). We only simulated faults in the

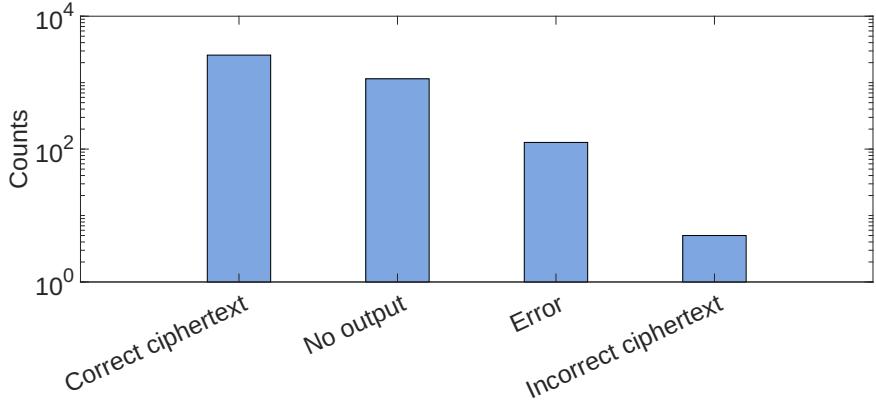


Figure 3.10: `aes-256-ctr` simulation results

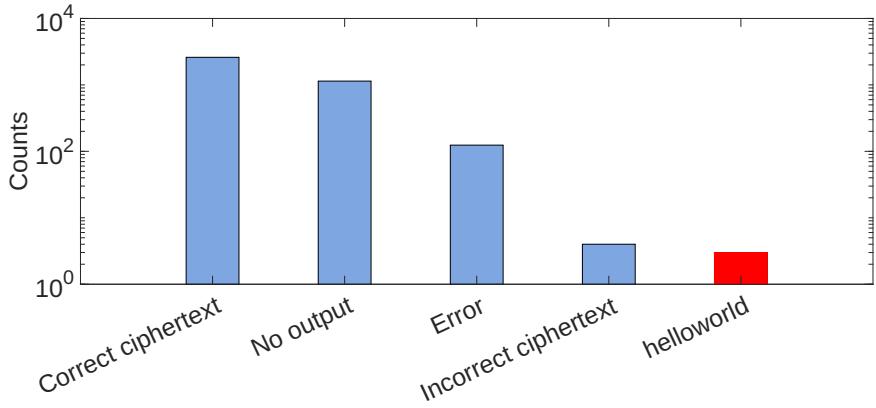


Figure 3.11: `aria-256-ctr` simulation results. Plaintext `helloworld` is revealed three times.

last 12 bits (which should be the same across all x86 machines the process is compiled for), thus, the LeapFrog gadgets should work across machines without the need for rescanning.

Post-Quantum Cryptography Schemes We use Open Quantum Safe (`liboqs version 0.11.1-dev`) library [194], an open source library for PQC algorithms, to find LeapFrog gadgets on FIPS standards using MFS tool.

One of the algorithms selected by NIST for standardization is CRYSTALS-Dilithium, which serves as a digital signature scheme providing post-quantum security guarantees. Dilithium relies on the hardness of structured lattice problems, such as the Learning With Errors (LWE) problem, which is believed to be intractable for quantum computers. Another prominent algorithm is FALCON, which offers smaller key sizes and signatures by employing

Table 3.6: 36 ciphers implemented in OpenSSL that are vulnerable to LeapFrog attack. Each given cipher reveals the plaintext fully or partially in the ciphertext due to skipped encryption steps.

Recovered	Cipher
helloworld	aria-128-cbc, aria-128-cfb, aria-128-cfb1 aria-128-cfb8, aria-128-ctr, aria-128-ofb aria192, aria-192-cbc, aria-192-cfb aria-192-cfb1, aria-192-cfb8, aria-192-ctr aria-192-ofb, aria256, aria-256-cbc aria-256-cfb, aria-256-cfb1, aria-256-cfb8 aria-256-ctr, aria-256-ofb, bf-ofb rc2-ofb, rc4, rc4-40
helloworld	bf-cfb, rc2-cfb
hdlmowor...	idea-cfb, idea-ofb
oworhell...	bf, bf-cbc, bf-ecb, blowfish
?rl#a?gy?...	chacha20, des-ed3-ofb, des-ed3-ofb, des-ofb

```

<enc_main>:
...
0x55555559c0c0: call  <opt_next>
0x55555559c0c5: test  %eax,%eax
0x55555559c0c7: je    <enc_main+0x1a0>
0x55555559c0c9: cmp   $0x1d,%eax
0x55555559c0cc: jg    <enc_main+0x178>
...
0x55555559c4b0: call  <opt_unknown>
0x55555559c4b5: lea   0x90(%rsp),%rsi
0x55555559c4bd: mov   %rax,%rdi
0x55555559c4c0: call  <opt_cipher>           ←
0x55555559c4c5: test  %eax,%eax
0x55555559c4c7: je    483a8 <enc_main+0x438>
0x55555559c4cd: mov   0x90(%rsp),%rbp
0x55555559c4d5: jmp   <enc_main+0x150>          ←
0x55555559c4da: nopw  0x0(%rax,%rax,1)
0x55555559c4e0: mov   0x84(%rsp),%r9d
...

```

Figure 3.12: LeapFrog gadget in OpenSSL command line tool resulting in encryption bypass in aria-128-cbc block cipher. The PC value that fault is injected into, $addr_{src}$, is highlighted in blue. The new value after the fault injected, $addr_{dest}$, is highlighted in red. The fault is injected during the execution of the function call highlighted in green.

the NTRU lattice, making it a competitive choice for constrained environments. Our analysis of these algorithms reveals that, despite their robust design against quantum attacks, they still exhibit vulnerabilities at the implementation level, susceptible to hardware fault injections like the LeapFrog used in Rowhammer-based exploits.

In digital signature schemes, we find gadgets that produce several failure modes in the Open Quantum Safe Library. The most critical error is a bypass of the signature verification. Note, that while we experimented with Post-Quantum encryption schemes, theoretically LeapFrog gadgets should work on classical encryption schemes as well.

The “Magic Number Mismatch” column in Table 3.7 highlights instances where the injected fault corrupts memory regions containing predefined magic numbers used for integrity checks. This mismatch signifies unintended memory corruption caused by the LeapFrog gadget, which can lead to unpredictable behavior or system crashes. According to Table 3.7, all signature schemes also contain gadgets for this failure mode, with Dilithium 3 containing the most number of LeapFrog gadgets.

Failures during key generation (“Key Gen. Fail”), signature generation (“Sig. Gen. Fail”), and signature verification (“Sig. Verif. Fail”) were also identified. Such failures can be exploited to disrupt normal cryptographic operations, resulting in denial-of-service (DoS) attacks or weakening cryptographic strength by producing invalid or insecure keys and signatures. All schemes contain this type of gadget.

The “Incorrect Verification” column denotes scenarios where invalid signatures are erroneously accepted as valid. This occurs when a LeapFrog gadget alters the control flow of the verification routine, enabling attackers to perform impersonation attacks by forging signatures that bypass standard validation checks.

Lastly, the “Verification Bypass” column in Table 3.7 highlights instances where the signature verification routine can be entirely circumvented using LeapFrog gadgets. This allows an attacker to craft an invalid signature and have it accepted as valid at the client’s end. By flipping bits in the Program Counter (PC) values using the LeapFrog within the client’s

memory space, the attacker effectively bypasses the signature verification routine. This vulnerability poses a significant security risk by enabling impersonation attacks and facilitating unauthorized access or actions within the system. Notably, Dilithium3 exhibits the highest number of LeapFrog gadgets for this threat, indicating a greater susceptibility to such attacks. An example of such a LeapFrog gadget in Dilithium is seen in Figure 3.13.

Table 3.7: Results from scans on the liboqs library, showing various issues encountered during signature operations for each digital signature scheme, along with the total number of assembly executions and candidate gadgets.

Scheme	# Inst.	Candidate Gadgets	Magic Num.	Key Gen. Fail	Sig. Gen. Fail	Sig. Verif. Fail	Incorrect Verif.	Bypass
Dilithium2	43474	17472	18	3	11	294	5	8
Dilithium3	42800	25010	36	6	22	654	10	20
Dilithium5	44171	18591	12	8	14	355	4	13
Falcon-512	60647	18641	18	16	50	106	4	16
Falcon-1024	60794	9360	8	8	22	40	2	7
Falcon-padded-512	60678	9396	8	9	21	41	2	6
Falcon-padded-1024	61128	9456	8	6	19	42	2	9
MAYO-1	43542	6924	8	3	4	33	2	8
MAYO-2	43533	6984	8	4	5	32	2	8
MAYO-3	46823	6996	8	3	4	35	2	6
MAYO-5	44302	6528	6	4	3	34	3	11
ML-DSA-44	43668	17616	0	3	1	21	1	5
ML-DSA-44-ipd	43338	8820	9	3	4	172	3	8
SPHINCS+-SHA2-128f	37109	8052	8	3	5	112	6	9
SPHINCS+-SHA2-128s	37379	8052	8	3	8	109	5	7
SPHINCS+-SHA2-192f	42143	8460	7	3	9	113	5	7
SPHINCS+-SHA2-192s	42535	8484	7	3	10	109	4	6
SPHINCS+-SHA2-256f	42437	8532	8	3	4	94	3	8
SPHINCS+-SHA2-256s	42758	8556	8	3	6	98	3	7
cross-rsdp-128-balanced	44024	10032	9	4	4	209	2	8

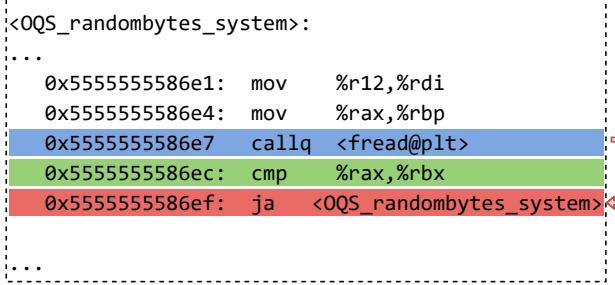


Figure 3.13: LeapFrog gadget detected in `liboqs` binary for Dilithium PQC Digital Signature Scheme. The PC value that fault is injected into, $addr_{src}$, is highlighted in blue. The new value after the fault injected, $addr_{dest}$, is highlighted in red. The fault is injected during the execution of the function call highlighted in green.

We find LeapFrog gadgets on FIPS 204 standard, also on other PQC digital signatures schemes, FALCON [58], MAYO [21], and CROSS [17].

Table 3.7 summarizes LeapFrog gadgets found in the `liboqs` library for different PQC digital signature schemes. Compared to Dilithium, ML-DSA, the implementation of the FIPS 204 standard, had fewer LeapFrog gadgets, suggesting that its implementation might be more resilient to the specific fault attacks we conducted. However, this does not imply immunity, as the gadgets found were still capable of bypassing critical functions. The relatively lower number of vulnerabilities in ML-DSA could also be attributed to its simpler structure, which reduces the surface area for potential control flow subversion attacks.

We also evaluated SPHINCS+, a hash-based signature scheme standardized in FIPS 205. SPHINCS+ offers a different security foundation, relying on the hardness of hash-based constructions rather than lattice problems. While this scheme is robust against certain classes of attacks, our analysis uncovered several LeapFrog gadgets capable of bypassing signature verification. This suggests that even though the algorithm itself is designed to withstand quantum and classical cryptanalytic attacks, practical vulnerabilities arise due to implementation flaws that allow Rowhammer-based attacks to alter execution paths. Interestingly, the number of LeapFrog gadgets identified in SPHINCS+ varied significantly based on its parameter set, with some configurations being more resilient than others. This highlights the importance of parameter selection in mitigating the risk of physical attacks.

SPHINCS+ has more gadgets compared to the FALCON-1024 configuration, but in some configurations, it has fewer gadgets than FALCON-512, another selected algorithm that is not standardized. Overall, our findings indicate that there are generally more LeapFrog gadgets that enable bypassing signature verification compared to those that can falsely verify an invalid signature, indicating higher feasibility for DoS attacks with lower security impact compared to impersonation attacks.

3.10 Conclusion

We analyzed the viability of a real-world DNN backdoor injection attack. Our backdoor attack scenario applies to deployed models by flipping a few bits in memory assisted by the Rowhammer. Our initial analysis performed on hardware showed that earlier proposals fall short in assuming a realistic fault injection model. We devised a new backdoor injection attack method that adopts a combination of trigger pattern generation and sparse and uniform weight optimization. In contrast to earlier proposals, our technique uses all layers and combines trigger pattern generation, target neuron selection, and fine-tuning model parameter weights in the same training loop. Since our approach targets the weight parameters uniformly, it is guaranteed that no more than one bit in a memory page is flipped. Further, we introduced new metrics to capture a realistic fault injection model. This new approach achieves a viable solution to target real-life deployments: on CIFAR10 (ResNet 18, 20, 32 models) and ImageNet (Resnet34 and 50 models) on real hardware by running the Rowhammer attack achieving Test Accuracy and Attack Success Rates as high as 92.95% and 95.26%, respectively. We also showed that our attack works on other architectures, such as VGG11 and VGG16. Finally, we evaluated the prominent defense techniques against our backdoor injection attack. We concluded that the proposed countermeasures are either not effective or introduce significant overhead in terms of time and storage.

Chapter 4

Scalable Generation and Detection of Spectre Gadgets

4.1 Motivation

The new era of microarchitectural attacks began with newly discovered Spectre [105] and Meltdown [119] attacks, which may be exploited to exfiltrate confidential information through microarchitectural channels during speculative and out-of-order executions. The Spectre attacks target vulnerable code patterns called gadgets, which leak information during speculatively executed instructions. While the initial variants of Spectre [105] exploit conditional and indirect branches, Koruyeh et al. [107] propose another Spectre variant by poisoning the entries in Return-Stack-Buffers (RSBs). Moreover, new Spectre-type attacks [36, 107] are implemented against the SGX environment and even remotely over the network [189]. These attacks show the applicability of Spectre attacks in the wild.

Unfortunately, chip vendors try to patch the leakages one-by-one with microcode updates rather than fixing the flaws by changing their hardware designs. Therefore, developers rely on automated malware analysis tools to eliminate mistakenly placed Spectre gadgets in their programs. The proposed detection tools mostly implement taint analysis [224] and

symbolic execution [72, 222] to identify potential gadgets in benign applications. However, the methods proposed so far are associated with two shortcomings: (1) the low number of Spectre gadgets prevents the comprehensive evaluation of the tools, (2) time consumption exponentially increases when the binary files become larger. Thus, there is a need for a robust and fast analysis tool that can automatically discover potential Spectre gadgets in large-scale commercial software.

Natural Language Processing (NLP) techniques are applied to automate challenging natural language and text processing tasks [172]. Later, NLP techniques have been used in the security domain, such as network traffic [173] and vulnerability analysis [180]. Such applications leverage word [140] or paragraph [113] embedding techniques to learn the vector representations of the text. The success of these techniques heavily depends on the large data sets, which ease training scalable and robust NLP models. However, for Spectre, for instance, the number of available gadgets is only 15, making it crucial to create new Spectre gadgets before building an NLP-based detection tool.

Generative Adversarial Networks (GANs) [65] are a type of generative models, which aim to produce new examples by learning the distribution of training instances in an adversarial setting. Since adversarial learning makes GANs more robust and applicable in real-world scenarios, GANs have become quite popular in recent years with applications ranging from generating images [156, 228] to text-to-image translation [181], etc. While the early applications of GANs focused on computer vision, implementing the same techniques in NLP tasks poses a challenge due to the lack of continuous space in the text. Various mathematical GAN-based techniques have been proposed to achieve better success in generating human-like sentences to overcome this obstacle [56, 74]. However, it is still unclear whether GANs can be implemented in the context of computer security to create application-specific code snippets. Additionally, each computer language has a different structure, semantics, and other features that make it more difficult to generate meaningful snippets for a specific application.

Neural vector embeddings [113, 140] used to obtain the vector representations of words have proven extremely useful in NLP applications. Such embedding techniques also enable one to perform vector operations in high dimensional space while preserving the meaningful relations between similar words. Typically, supervised techniques apply word embedding tools as an initial step to obtain the vector embedding of each token and then build a supervised model on top. For instance, BERT [51] was proposed by the Google AI team, which learns the relations between different words in a sentence by applying a self-attention mechanism [220]. BERT has exhibited superior performance compared to previous techniques [139, 196] when combined with bi-directional learning. Furthermore, the attention mechanism improves GPU utilization while learning long sequences more efficiently. Recently, BERT-like architectures are shown to be capable of modeling high-level programming languages [57, 111]. However, it is still unclear whether it will be effective to model a low-level programming language, such as Assembly language, and help build more robust malware detection tools, which is the goal of this work.

Our Contributions Our contributions are twofold. First, we increase the diversity of Spectre gadgets with the mutational fuzzing technique. We start with 15 examples [104] and produce 1 million gadgets by introducing various instructions and operands to the existing gadgets. Then, we propose a GAN-based tool, namely, SpectreGAN, which learns the distribution of 1 million Spectre gadgets to generate new gadgets with high accuracy. The generated gadgets are evaluated from both semantic and microarchitectural aspects to verify their diversity and quality. Furthermore, we introduce novel gadgets that are not detected by state-of-the-art detection tools.

In the second part, we introduce FastSpec, a high dimensional neural embedding based detection technique derived from BERT, to obtain a highly accurate and fast classifier for Spectre gadgets. We train FastSpec with generated gadgets and achieve a 0.998 Area Under the Curve (AUC) score for OpenSSL libraries in the test phase. Further, we apply FastSpec on Phoronix benchmark tests to show that FastSpec outperforms taint analysis-based and

symbolic execution-based detection tools as well as significantly decreases the analysis time.

In summary,

- We extend 15 base Spectre examples to 1 million gadgets by applying a mutational fuzzing technique,
- We propose SpectreGAN which leverages conditional GANs to create new Spectre gadgets by learning the distribution of existing Spectre gadgets in a scalable way,
- We show that both mutational fuzzing and SpectreGAN create diverse and novel gadgets which are not detected by *oo7* and *Spectector* tools,
- We introduce FastSpec, which is based on supervised neural word embeddings to identify the potential gadgets in benign applications orders of magnitude faster than rule-based methods.

4.2 Related Works

4.2.1 Spectre attacks and detectors

Spectre Variations and Covert Channels In the first Spectre study [105], two variants were introduced. While Spectre-V1 exploits the conditional branch prediction mechanism when a bound check is present, Spectre-V2 manipulates the indirect branch predictions to leak the secret. Next, researchers discovered new variants of Spectre-based attacks. For instance, a variant of Spectre focuses on poisoning Return-Stack-Buffer (RSB) entries with the desired malicious return addresses [107, 131]. Another variant of Spectre called Speculative Store Bypass [82] takes advantage of the memory disambiguator’s prediction to create leakage. Traditionally, secrets are leaked through cache timing differences. Then, researchers showed that there are also other covert channels to measure the time difference: namely using network latency [189], port contention [22], or control flow hijack attack based on return-oriented programming [133] to leak secret data.

Defenses against Spectre There are various detection methods for speculative execution attacks. Taint analysis is used in *oo7* [224] software tool to detect leakages. As an alternative way, the taint analysis is implemented in the hardware context to stop the speculative execution for secret dependent data [188, 245]. The second method relies on symbolic execution analysis. Spectector [72] symbolically executes the programs where the conditional branches are treated as mispredicted. Furthermore, SpecuSym [76] and KleeSpectre [222] aim to model cache usage with symbolic execution to detect speculative interference, which is based on Klee symbolic execution engine. Following a different approach, Speculator [132] collects performance counter values to detect mispredicted branches and speculative execution domain. Finally, Specfuzz [159] leverages a fuzzing strategy to test functions with diverse set of inputs. Then, the tool analyzes the control flow paths and determines the most likely vulnerable code snippets against speculative execution attacks.

4.2.2 Binary Analysis with Embedding

Binary analysis is one of the methods to analyze the security of a program. The analysis can be performed dynamically [150] by observing the binary code running in the system. Alternatively, the binary can also be analyzed statically [193]. NLP techniques have been applied to binary analysis in recent years. Mostly, the studies leverage the aforementioned techniques to embed Assembly instructions and registers into a vector space. The most common usage of NLP in the binary analysis is to find the similarities between files. Asm2Vec [52] leverages a modified version of the PV-DM model to solve the obfuscation and optimization issues in a clone search. Zuo et al. [249] and Redmond et al. [180] solve the binary similarity problem by NLP techniques when the same file is compiled in different architectures. SAFE [135] proposes a combination of skip-gram and RNN self-attention models to learn the embeddings of the functions from binary files to find the similarities.

4.2.3 GAN-based Text Generation

The first applications of GANs were mostly applied to computer vision to create new images such as human faces [97,98], photo blending [233], video generation [221], and so on. However, text generation is a more challenging task since it is more difficult to evaluate the performance of the outputs. An application [117] of GANs is in the dialogue generation, where adversarial learning and reinforcement are applied together. SeqGAN [246] introduces gradient policy update with Monte Carlo search. LeakGAN [75] implements a modified policy gradient method to increase the usage of word-based features in adversarial learning. RelGAN [154] applies Gumbel-Softmax relaxation for training GANs as an alternative method to gradient policy update. SentiGAN [226] proposes multiple generators to focus on several sentiment labels with one multi-class generator. However, to the best of our knowledge, the literature lacks GANs applied to the Assembly code generation. To fill this literature gap, we propose SpectreGAN in Section 4.3.2.

4.3 SpectreGAN: Spectre Gadget Generation

We propose both mutational fuzzing and GAN-based gadget generation techniques to create novel and diverse gadgets. In the following sections, details of both techniques and the diversity analysis of the gadgets are given:

4.3.1 Gadget Generation via Fuzzing

We begin with fuzzing techniques to extend the base gadgets to create an extensive data set consists of a million Spectre gadgets in four steps.

- **Step 1: Initial Data Set** There are 15 Spectre-V1 gadgets written in C by Kocher [104] and two modified examples introduced by *Spectector* [72]. For each example, a different attacker code is written to leak the entire secret data completely in a reasonable time.

Algorithm 2: Gadget generation using mutational fuzzing

Input: An Assembly function A , a set of instructions \mathbb{I}_b and sets of registers \mathbb{R}_b for different sizes of b

Output: A mutated Assembly function A'

```
1  $\mathbb{G} := \mathbb{R}_b \mapsto \mathbb{I}_b$ 
2  $A' = A$ 
3  $MaxOffset = length(A)$ 
4 for  $1:Diversity$  do
5   for  $Offset=1:MaxOffset$  do
6     for  $1:Offset$  do
7        $i_b \leftarrow random(\mathbb{I})$ 
8        $r_b \leftarrow random(\mathbb{R}_b | \mathbb{G})$ 
9        $l \leftarrow random(0 : length(A'))$ 
10      Insert( $\{i_b | r_b\}$ ,  $A'$ ,  $l$ )
11    end
12    Test boundary check( $A'$ )
13    Test Spectre leakage( $A'$ )
14  end
15 end
```

- **Step 2: Compiler variants and optimization levels** Since our target data set is in assembly code format, each Spectre gadget written in C is compiled into x86 assembly functions using different compilers. We compiled each example with *GCC*, *clang*, and *icc* compilers using *-O0* and *-O2* optimization flags. Therefore, we obtain 6 different assembly functions from each C function with AT&T syntax.
- **Step 3: Mutational fuzzing based generation**

We generated new samples with an approach inspired by mutation-based fuzzing technique [197] as introduced in Algorithm 2. Our mutation operator is the insertion of random assembly instructions with random operands. For an assembly function A with length L , we create a mutated assembly function A' . We set a limit on the number of generated samples per assembly function A for each $Offset$ value, denoted as $Diversity$. We choose a random instruction i_b from the instruction set \mathbb{I} , and depending on the instruction format of i_b ; we choose random operands r_b , which are compatible with the instruction in terms of bit size, b . After proper instruction-operand selection, we choose a random position l in A' and insert $\{i_b | r_b\}$ into that location. We repeat the insertion process until we reach the $Offset$

value. The randomly inserted instruction and register list are given in Appendix A.2.

- **Step 4: Verification of the generated gadgets**

Finally, A' is tested whether it is still a Spectre-V1 gadget or not. There are two verification tests that are applied to the generated functions.

The first verification test is applied to make sure that the function still has the proper array boundary-check for given user inputs. Since random instructions are inserted in random locations in the gadget, a new instruction may alter the flags whose value is checked by the original conditional jump. Once the flags are broken, the secret may be leaked without any speculative execution. To test this case, the PoC Spectre-V1 attacker code [105] is modified to supply only out-of-bounds inputs to A' , which prevents mistraining the branch predictor. If the secret bytes in the PoC code are still leaked, we conclude that the candidate gadget is broken and exclude it from the pool.

If a generated function A' passes from the first test, we apply the PoC Spectre-V1 attack to the gadget and exclude it if it does not leak the secret data through speculative execution. Additionally, the verification code is modified based on Kocher’s examples since each example gadget leaks the secret in a different way. For instance, 4th example shifts the user input by 1, which affects the leakage mapping in the cache. Therefore, we modified the PoC code to compile it with the generated gadgets together to leak the whole secret. This process is repeated for each example in Kocher’s gadget dataset [104], which yields 16 different verification codes. The secret in the gadgets is only decoded via implementing the Flush+Reload technique. Other microarchitectural side-channels are not in the scope of the verification phase.

Other Spectre variants such as SmotherSpectre [22] and NetSpectre [189] are not in our scope. Hence, the generated gadgets that potentially include SmotherSpectre and NetSpectre variants are not verified with other side-channel attacks. Our verification procedure only guarantees that the extracted gadgets leak secret information through cache side-channel

attacks. The verification method can be adjusted to other Spectre variants, which is explained further in Section 4.5.

At the end of the fuzzing-based generation, we obtained a data set of almost 1.1 million Spectre gadgets¹. The overall success rate of the fuzzing technique is 5% out of compiled gadgets. The generated gadgets are used to train SpectreGAN in the next section.

4.3.2 SpectreGAN: Assembly Code Generation with GANs

We introduce SpectreGAN, which learns the fuzzing generated gadgets in an unsupervised way and generates new Spectre-V1 variants from existing assembly language samples. The purpose of SpectreGAN is to develop an intelligent way of creating assembly functions instead of randomly inserting instructions and operands. Hence, the low success rate of gadget generation in the fuzzing technique can be improved further with GANs. To the best of our knowledge, the literature lacks GANs applied to the assembly code generation. To fill this literature gap, we propose SpectreGAN in Section 4.3.2.

We build SpectreGAN based on the MaskGAN model, with 1.1 million examples generated in Section 4.3. Since MaskGAN is originally designed for text generation, we modify the MaskGAN architecture to train SpectreGAN on assembly language. Finally, we evaluate the performance of SpectreGAN and discuss challenges in assembly code generation.

4.3.2.1 SpectreGAN Architecture

SpectreGAN has a generator model that learns and generates x86 assembly functions and a discriminator model that gives feedback to the generator model by classifying the generated samples as real or fake as depicted in Figure 4.1.

Generator The generator model consists of encoder-decoder architecture (seq2seq) [196] which is composed of two-layer stacked LSTM units. Firstly, the input assembly functions are

¹The attacker codes for each example, the entire data set, SpectreGAN, and FastSpec code are available at <https://github.com/vernamlab/FastSpec>

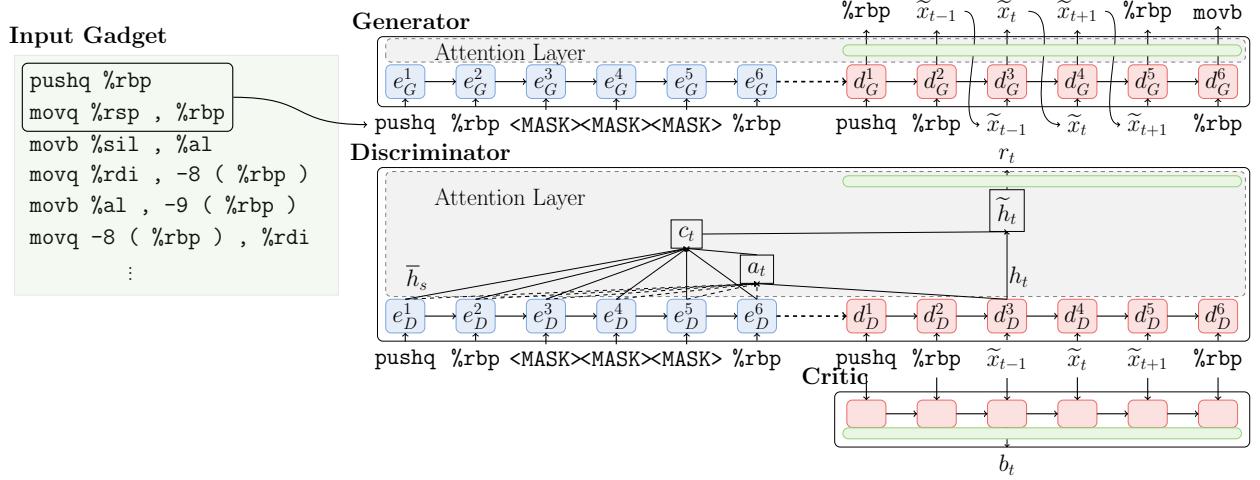


Figure 4.1: SpectreGAN architecture. Blue and red boxes represent the encoder and decoder LSTM units, respectively. Green boxes represent the softmax layers. The listed assembly function (AT&T format) on the left is fed to the models after the tokenization process. The critic model and the decoder part of the discriminator get the same sequence of instructions in the adversarial training.

converted to a sequence of tokens $T' = \{x'_1, \dots, x'_N\}$ where each token represents an instruction, register, parenthesis, comma, intermediate value or label. SpectreGAN is conditionally trained with each sequence of tokens where a masking vector $m = (m_1, \dots, m_N)$ with elements $m_t \in \{0, 1\}$ is generated. The masking rate of m is determined as $r_m = \frac{1}{N} \sum_{t=1}^N m_t$. $m(T')$ is the modified sequence where x'_t is replaced with <MASK> token for the corresponding positions of $m_t = 1$. Both T' and $m(T')$ are converted into the lists of vectors $T = \{x_1, \dots, x_N\}$ and $m(T)$ by a lookup in a randomly initialized embedding matrix of size $V \times H$, where V and H are the vocabulary size and embedding vector dimension, respectively. In order to learn the masked tokens, T and $m(T)$ are fed into the encoder LSTM units of the generator model. Each encoder unit outputs a hidden state \bar{h}_s which is also given as an input to the next encoder unit. The last encoder unit (e_G^6 in Figure 4.1) produces the final hidden state which encapsulates the information learned from all assembly tokens.

The decoder state is initialized with the encoder's final hidden state, and the decoder LSTM units are fed with $m(T)$ at each iteration. To calculate the hidden state \tilde{h}_t of each decoder unit, the attention mechanism output and the current state of the decoder h_t are

combined. The attention mechanism reduces the information bottleneck between encoder and decoder and eases the training [15] on long token sequences in assembly function data set. The attention mechanism is implemented exactly same for both generator and discriminator model which is illustrated in the discriminator part in Figure 4.1. The alignment score vector a_t is calculated as:

$$a_t(s) = \frac{e^{h_t^\top \bar{h}_s}}{\sum_{s'=1}^N e^{h_t^\top \bar{h}_{s'}}}, \quad (4.1)$$

where a_t describes the weights of \bar{h}_s , for a token x'_t at time step t , where $h_t^\top \bar{h}_s$ is the score value between the token x'_t and T' . This forces decoder to consider the relation between each instruction, register, label and other tokens before generating a new token. The context vector c_t is calculated as the weighted sum of \bar{h}_s as follows:

$$c_t = \sum_{s'=1}^N a_t(s) \bar{h}_{s'}. \quad (4.2)$$

For a context vector, c_t , the final attention-based hidden state, \tilde{h}_t , is obtained by a fully connected layer with hyperbolic tangent activation function,

$$\tilde{h}_t = \tanh(W_c[c_t; h_t]), \quad (4.3)$$

where $[c_t; h_t]$ is the concatenation of c_t and h_t with the trainable weights W_c . The output list of tokens $\tilde{T} = (\tilde{x}_1, \dots, \tilde{x}_N)$ is then generated by filling the masked positions for $m(T')$ where $m_t = 1$. The probability distribution $p(y_t | y_{1:t-1}, x_t)$ is calculated as,

$$p(y_t | y_{1:t-1}, x_t) = \frac{e^{W_s \tilde{h}_t}}{\sum e^{W_s \tilde{h}_t}}, \quad (4.4)$$

where y_t is the output token and attention-based hidden state \tilde{h}_t is fed into the softmax layer which is represented by the green boxes in Figure 4.1. It is important to note that the softmax layer is modified to introduce a randomness at the output of the decoder by a

sampling operation. The predicted token is selected based on the probability distribution of vocabulary, *i.e.* if a token has a probability of 0.3, it will be selected with a 30% chance. This prevents the selection of the token with the highest probability every time. Hence, at each run the predicted token would be different which increases the diversity in the generated gadgets.

Discriminator The discriminator model has a very similar architecture to the generator model. The encoder and decoder units in the discriminator model are again two-layer stacked LSTM units. The embedding vectors $m(T)$ of tokens $m(T')$, where we replace x'_t with <MASK> when $m_t = 1$, are fed into the encoder. The hidden vector encodings \bar{h}_s and the encoder's final state are given to the decoder.

The LSTM units in the decoder are initialized with the final hidden state of the encoder and \bar{h}_s is given to the attention layer. The list of tokens \tilde{T} which represents the generated assembly function by the generator model is fed into the decoder LSTM unit with *teacher forcing*. The previous calculations for $a_t(s)$, c_t and \tilde{h}_t stated in Equation 4.1, 4.2, and 4.3 are valid for the attention layer in the discriminator model as well. The attention-based state value \tilde{h}_t is fed through the softmax layer which outputs only one value at each time step t ,

$$p_D(\tilde{x}_t = x_t^{real} | \tilde{T}) = \frac{e^{W_s \tilde{h}_t}}{\sum e^{W_s \tilde{h}_t}}, \quad (4.5)$$

which is the probability of being a real target token x_t^{real} .

SpectreGAN has one more model apart from the generator and the discriminator models, which is called the critic model, and it has only one two-layer stacked LSTM unit. The critic model is initialized with zero states and gets the same input \tilde{T} with the decoder. The output of the LSTM unit at each time step t is given to the softmax layer, and we obtain

$$p_C(\tilde{x}_t = x_t^{real} | \tilde{T}) = \frac{e^{W_b h_t}}{\sum e^{W_b h_t}}, \quad (4.6)$$

which is an estimated version of p_D . The purpose of introducing a critic model for probability

estimation will be explained in Section 4.3.2.2.

4.3.2.2 Training

The training procedure consists of two main phases namely, pre-training and adversarial training.

Pre-training phase The generator model is first trained with maximum likelihood estimation. The real token sequence T' and masked version $m(T')$ are fed into the generator model's encoder. Only the real token sequence T' is fed into the decoder using *teacher forcing* in the pre-training. The training maximizes the log-probability of generated tokens, \tilde{x}_t given the real tokens, x'_t , where $m_t = 1$. Therefore, the pre-training objective is

$$\frac{1}{N} \sum_{t=1}^N \log p(m(\tilde{x}_t) | m(x'_t)), \quad (4.7)$$

where $p(m(\tilde{x}_t) | m(x'_t))$ is calculated only for the masked positions. The masked pre-training objective ensures that the model is trained for a *Close* task [203].

Adversarial training phase The second phase is adversarial training, where the generator and the discriminator are trained with the GAN framework. Since the generator model has a sampling operation from the probability distribution stated in Equation 4.4, the overall GAN framework is not differentiable. We utilize the policy gradients to train the generator model, as described in the previous works [56, 246].

The reward r_t for a generated token \tilde{x}_t is calculated as the logarithm of $p_D(\tilde{x}_t = x_t^{real} | \tilde{T})$. The aim of the generator model is to maximize the total discounted rewards $R_t = m(\sum_{s=t}^N \gamma^s r_s)$ for the fake samples, where γ is the discount factor. Therefore, for each token, the generator is updated with the gradient in Equation 4.8 using the REINFORCE algorithm, where $b_t = \log p_C(\tilde{x}_t = x_t^{real} | \tilde{T})$ is the baseline rewards by the critic model. Subtracting b_t from R_t

helps reducing the variance of the gradient [56].

$$\nabla_{\theta} \mathbb{E}_G[R_t] = (R_t - b_t) \nabla_{\theta} \log G_{\theta}(\tilde{x}_t) \quad (4.8)$$

To train the discriminator model, both real sequence T and fake sequence \tilde{T} are fed into the discriminator. Then, the model parameters are updated such that $\log p_D(\tilde{x}_t = x_t^{real} | \tilde{T})$ is minimized and $\log p_D(x_t = x_t^{real} | T)$ is maximized using maximum log-likelihood estimation.

4.3.2.3 Tokenization and Training Parameters

Firstly, we pre-process the fuzzing generated data set to convert the assembly functions into sequences of tokens, $T' = (x'_1, \dots, x'_N)$. We keep commas, parenthesis, immediate values, labels, instruction and register names as separate tokens. To decrease the complexity, we reduce the tokens' vocabulary size and simplify the labels in each function so that the total number of different labels is minimum. The tokenization process converts the instruction "movq (%rax), %rdx" into the list ["movq", "(", "%rax", ")", ",", "%rdx"] where each element of the list is a token x'_t . Hence, each token list $T' = \{x'_1, \dots, x'_N\}$ represents an assembly function in the data set.

The masking vector has two different roles in the training. While a random masking vector $m = (m_1, \dots, m_N)$ is initialized for the pre-training, we generate m as a contiguous block with a random starting position in the adversarial training. In both training phases, the first token's mask is always selected as $m_1 = 0$, meaning that the first token given to the model is always real. The masking rate, r_m determines the ratio of masked tokens in an assembly function whose effect on code generation is analyzed further in Section 4.3.2.4.

SpectreGAN is configured with the embedding vector size of $d = 64$, generator learning rate of $\eta_G = 5 \times 10^{-4}$, discriminator learning rate of $\eta_D = 5 \times 10^{-3}$, critic learning rate of $\eta_C = 5 \times 10^{-7}$ and discount rate of $\gamma = 0.89$ based on the MaskGAN implementation [56]. We select the sequences with a maximum length of 250 tokens, building the vocabulary with a

size of $V = 419$. We separate 10% of the data set for model validation. SpectreGAN is trained with a batch size of 100 on NVIDIA GeForce GTX 1080 Ti until the validation perplexity converges in Figure 4.2. The pre-training lasts about 50 hours, while the adversarial training phase takes around 30 hours.

4.3.2.4 Evaluation

SpectreGAN is based on learning masked tokens with the surrounding tokens. The masking rate is not a fixed value, which is determined based on the context. Since SpectreGAN is the first study to train on Assembly functions, the masking rate choice is of utmost importance to generate high-quality gadgets. Typically, NLP-based generation techniques are evaluated with their associated perplexity score, which indicates how well the model predicts a token. Hence, we evaluate the performance of SpectreGAN with various masking sizes and their perplexity scores. In Figure 4.2, the perplexity converges with the increasing number of training steps, which means the tokens are predicted with a higher accuracy towards the end of the training. SpectreGAN achieves lower perplexity with higher masking rates, which indicates that higher masking rates are more preferable for SpectreGAN.

Even though the higher masking rates yield lower perplexity and assembly functions of high quality in terms of token probabilities, our purpose is to create functions which behave as Spectre gadgets. Therefore, as a second test, we generated 100,000 gadgets for 5 different masking rates. Next, we compiled our gadgets with the *GCC* compiler and then tested them with all the attacker code to verify their secret leakage. When SpectreGAN is trained with a masking rate of 0.3, the success rate of gadgets increases by up to 72%. Interestingly, the success rate drops for other masking rates, demonstrating the importance of masking rate choice. In total, 70,000 gadgets are generated with a masking rate of 0.3 to evaluate the performance of SpectreGAN in terms of gadget diversity in Section 4.3.3.

To illustrate an example of the generated samples, we fed the gadget in Listing 4.1 to SpectreGAN and generated a new gadget in Listing 4.2. We demonstrate that SpectreGAN

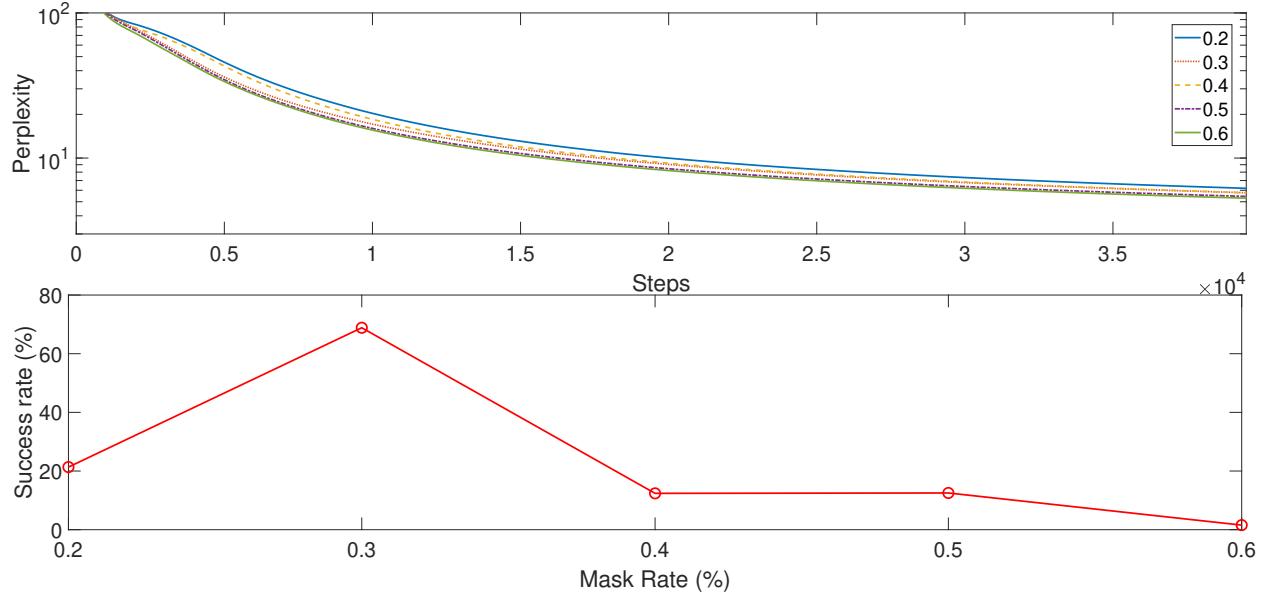


Figure 4.2: (Above) The validation perplexity decreases at each training step and converges for all r_m . (Below) Spectre gadget success rates are evaluated when different masking rates are used to train SpectreGAN. Spectre gadget success rate shows the percentage of gadgets out of compiled functions.

is capable of generating realistic assembly code snippets by inserting, removing, or replacing the instructions, registers, and labels. In the Listing 4.2, the lines that start with the instructions written with red color are generated by SpectreGAN, and they correspond to the masked portion of Spectre-V1 gadget given in Listing 4.1.

```

1 victim_function:
2 .cfi_startproc
3 movl size(%rip),%eax
4 cmpq %rdi,%rax
5 jbe .L0
6 leaq array1(%rip),%rax
7 movzbl (%rdi,%rax),%eax
8 ror $1,%rsi
9 shlq $9,%rax
10 leaq array2(%rip),%rcx
11 movss %xmm8,%xmm4
12 movb (%rax,%rcx),%al
13 andb %al,temp(%rip)
14 movd %xmm1,%r14d
15 test %r15,%rcx
16 sbbl %r13d,%r9d
17 .L0:
18 retq
19 cmovll %r8d,%r10d
20 .cfi_endproc

```

Listing 4.1: Input Spectre-V1 gadget

```

1 victim_function:
2 .cfi_startproc
3 movl size(%rip),%eax
4 cmpq %rdi,%rax
5 jbe .L0
6 leaq array1(%rip),%rax
7 movzbl (%rdi,%rax),%eax
8 ror $1,%rsi
9 shlq $9,%rax
10 movb array2(%rdi),%al
11 andb %al,temp(%rip)
12 .L1:
13 andb %r13b,%al
14 movb array2(%rax),%al
15 andb %al,temp(%rip)
16 sbbl %r13d,%r9d
17 .L0:
18 retq
19 cmovll %r8d,%r10d
20 .cfi_endproc

```

Listing 4.2: Generated gadget

4.3.3 Diversity and Quality Analysis of Generated Gadgets

In total, 1.2 million gadgets are generated by the mutational fuzzing technique and SpectreGAN. Since the gadgets are derived from existing examples, it is crucial to analyze their diversity and quality. The diversity is measured by syntactic analysis, e.g., counting the number of unique n-grams in gadgets. For the quality metric, we monitor performance counters while the gadgets are executed. 5000 gadgets are randomly selected from each gadget generation technique to perform syntactic and microarchitectural analysis. Furthermore, novel gadgets that are not detected by *oo7* [224] and *Spectector* [72] tools are given to show that our gadget generation techniques produce meaningful Spectre-V1 gadgets.

4.3.3.1 Syntactic Analysis

In NLP applications, the diversity of the generated texts is evaluated by counting the number of unique n-grams. The most common metrics for the text diversity are perplexity and BLEU scores that are calculated based on the probabilistic occurrences of n-grams in a sequence. The higher number of n-grams indicates that an NLP model learns the data set

distribution efficiently and produces new sequences with high diversity. However, both scores are obtained during the training phase; thus, making it impossible to evaluate the fuzzing generated gadgets since there is no training phase. Instead, we conduct diversity analysis by counting the unique n-grams introduced by fuzzing and SpectreGAN methods after all the gadgets are generated.

The number of unique n-grams in generated gadgets is compared with 17 base examples in Table 4.1. The unique n-grams are calculated as follows: First, unique n-grams produced by fuzzing are identified and stored in a list. Then, additional unique n-grams introduced by SpectreGAN are noted. Therefore, the unique n-grams generated by SpectreGAN in Table 4.1 represent the number of n-grams introduced by SpectreGAN, excluding fuzzing generated n-grams.

Table 4.1: Table shows the number of unique n-grams for base gadgets and generated gadgets by fuzzing and SpectreGAN methods. In the last column the total number of unique n-grams are given as well as the increase factor that improves with the increasing n-grams.

n	Base	Fuzzing	SpectreGAN	Total
2	2069	15,448	7,462	22,910 ($\times 11$)
3	3349	181,606	91,851	273,457 ($\times 82$)
4	4161	639,608	460,317	1,099,925 ($\times 264$)
5	4747	998,279	921,519	1,919,798 ($\times 404$)

In total, the number of unique bigrams (2-grams) is increased to 22,910 from 2,069, which is more than 10 times raise. While new instructions and registers added by fuzzing improve the gadgets' diversity, SpectreGAN contributes to the gadget diversity by producing unique perturbations. Since the instruction diversity increases drastically compared to base gadgets, the unique 5-grams reach up to almost 2 million, 400 times higher than the base gadgets. The results show that both fuzzing and SpectreGAN span the diversity in the generated gadgets. High diversity in the gadget data set also results in microarchitectural behavior diversity as well as new Spectre-V1 gadgets that were not previously considered during the design process of previous detection mechanisms.

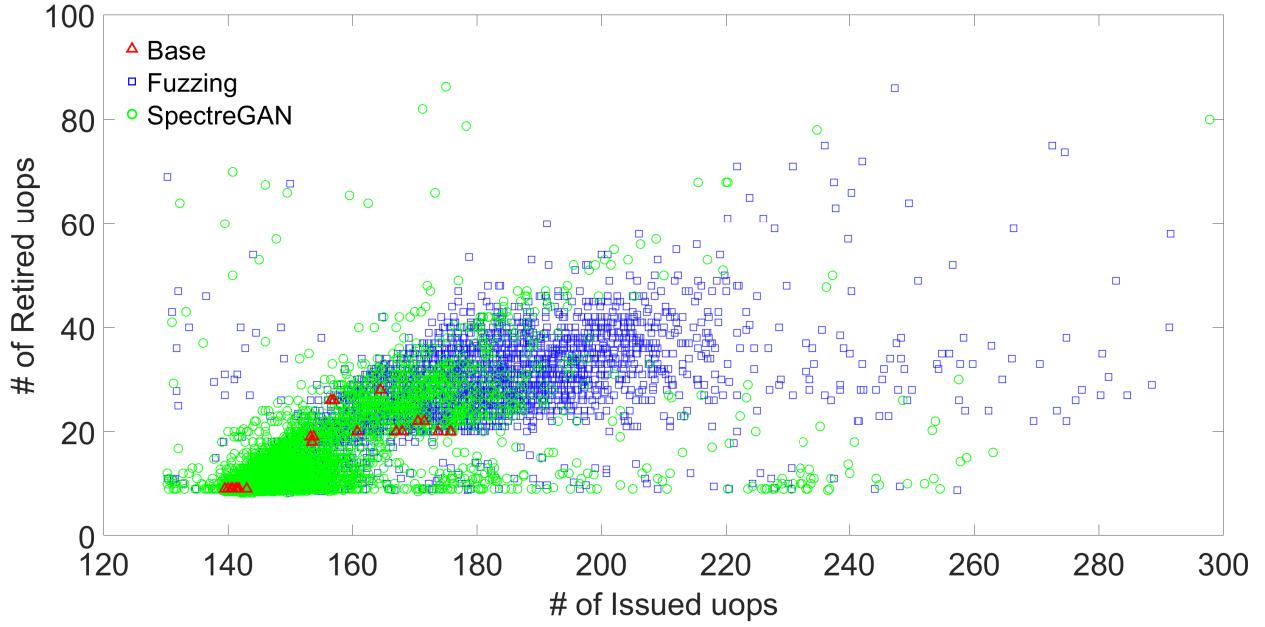


Figure 4.3: The distribution of base (red-triangle), fuzzing generated (blue-square) and SpectreGAN generated (green-circle) gadgets is given for issued and retired μ ops counters.

Both SpectreGAN and fuzzing techniques generate diverse set of gadgets in Haswell architecture.

4.3.3.2 Microarchitectural Analysis

Another purpose of gadget generation is to introduce new instructions and operands to create high-quality gadgets. To assess the quality of the gadgets, we analyze gadgets' microarchitectural characteristics. The first challenge is to examine the effects of instructions in the transient domain since they are not visible in the architectural state. After carefully analyzing the performance counters for Haswell architecture, we determined that two counters, namely, *uops_issued : any* and *uops_retired : any* give an insight into gadgets' microarchitectural behavior. *uops_issued : any* counter is incremented every time a μ op is issued, which counts both speculative and non-speculative μ ops. On the other hand, *uops_retired : any* counter only counts the executed and committed μ ops, which automatically excludes speculatively executed μ ops.

The performance counter distribution of generated gadgets and base gadgets are given in Figure 4.3. The gadget quality is measured by the number of instructions in the transient domain after a gadget passes the verification step. The exploitable gadgets in the commercial

software have many instructions that are speculatively executed until the secret is leaked. If our detection tool in Section 4.4 is only trained with simple gadgets from Kocher’s examples, the success rate would be low in large-scale software binaries. Moreover, the gadgets that are detected in the case studies are very similar to the generated gadgets which have more instructions in the transient domain. A similar observation is also shared in [225], where the authors claim that Spectre gadgets have up to 150 instructions between the conditional branch and speculative memory access in the detected gadgets. Since our aim is to create realistic gadgets by inserting various instructions, we assume that gadget quality increases in parallel when a gadget is close to the x-axis and far from the y-axis.

It is more likely to obtain high-quality gadgets with fuzzing method as new instructions and operands are randomly added. On the other hand, SpectreGAN learns the essential structure of the fuzzing generated gadgets, which yields almost the same number of samples close to the x-axis in Figure 4.3. Moreover, the advantage of SpectreGAN is to automate the creation of gadgets with a higher accuracy (72%) compared to the fuzzing technique (5%).

4.3.3.3 Detection Analysis

Even though the microarchitectural and syntactic analyses show that fuzzing and SpectreGAN can produce diverse and high-quality sets of gadgets, we aim to enable a comprehensive evaluation of detection tools and determine the most interesting gadgets in our data set. For this reason, the generated gadgets are fed into *Spectector* [72] and *oo7* [224] tools to determine the novelty of the gadgets.

oo7 tool leverages taint analysis to detect Spectre-V1 gadgets. It is based on the Binary Analysis Platform (BAP) [48] which forwards taint propagation along all possible paths after a conditional branch is encountered. *oo7*² is built on a set of hand-written rules which cover the existing examples by Kocher [104]. Although our data set size is 1.2 million, we have selected 100,000 samples from each gadget example uniformly random due to the immense

²<https://gitlab.com/igoto/spectre-detector>

time consumption of *oo7* (150 hours for 100K gadgets), which achieves a 94% detection rate.

```

1 void victim_function(size_t x){
2     if(global_condition)
3         x = 0;
4     if(x < size)
5         temp &= array2[array1[x] * 512];
6 }
```

Listing 4.3: CMov gadget: An example Spectre gadget in C format. When it is compiled with *gcc-7.5 -O2* optimization level, CMovcc gadget bypasses *oo7* tool. The generated assembly version is given in Appendix A.1.

Interestingly, specific gadget types from both fuzzing and SpectreGAN are not caught by *oo7*. When a gadget contains *cmove* or *xchg* or *set* instruction and its variants, it is not identified as a Spectre gadget. Hence, we introduce these gadgets as novel Spectre-V1 gadgets listed in Listing 4.3 and Listing 4.4. Their corresponding assembly snippets are also given in Appendix A.1.

```

1 size_t prev = 0xff;
2 void victim_function(size_t x) {
3     if (prev < size)
4         temp &= array2[array1[prev] * 512];
5     prev = x;
6 }
```

Listing 4.4: XCHG gadget: When a past value, that is controlled by the attacker, is used to leak the secret in the Spectre gadget, *oo7* cannot detect the XCHG gadget. This example show that control-flow graph extraction is not efficiently implemented in *oo7* tool.

We identified two potential issues of static taint analysis method in *oo7* tool. First, if a portion of a tainted variable is modified by an instruction such as *cmove* or *set*, the tainted variable is not tracked by the tool. However, an attacker still controls the remaining portion

of the variable, which makes it possible to leak the secret from memory. In some cases, the implementation of static taint analysis is not sufficiently accurate to track partially modified tainted variables due to under-tainting. Secondly, the tainted variables are not tracked between the iterations of a loop. If an old attacker-controlled variable is used to access the secret, *oo7* tool is not able to taint the old variable between the iterations of a *for* loop. Hence, any old attacker-controlled variable can be used to bypass the tool. This shows that control flow graphs of multiple iterations may not be extracted correctly by *oo7*. Both weaknesses show that hand-written rules do not generalize well for Spectre gadget detection when new Spectre-V1 gadgets are discovered.

Spectector [72] makes use of a symbolic execution technique to detect the potential Spectre-V1 gadgets. For each assembly file, *Spectector* is adjusted to track 25 symbolic paths of at most 5000 instructions each, with a global timeout of 30 minutes. The remaining parameters are kept as default.

First, we eliminate the gadgets that include unsupported instructions as these gadgets are never detected by *Spectector*. When we analyze the remaining gadgets, 1% of the gadgets are not detected successfully. Then, undetected gadgets are examined to determine novel gadgets.

We determined two issues in the *Spectector* tool. The first issue is related to the barrier instructions. Even though *lfence*, *sfence* and *mfence* instructions have different purposes, the tool treats them as equal instructions. For instance, if an *sfence* instruction is present after the conditional branch, the tool classifies the gadget as safe. However, *sfence* instruction has no effect on the load operation so, the gadget still leaks the secret. Hence, *Spectector*'s modeling of fences does not distinguish the differences between various x86 fence instructions. The second issue is about 8-bit registers in which a partial information of the elements in *array[x]* is stored. When 8-bit registers are used to modify the elements in Listing 4.5, *Spectector* is no longer able to detect the gadgets. This second issue is also mentioned in [72], i.e., sub-registers are currently not supported by the tool. Overall, these issues are

due to the problems in the translation from x86 assembly into Spectector’s intermediate language.

We show that our large-scale diverse gadget data set establishes a ground truth to evaluate the detection tools accurately. As shown in the case studies on *Spectector* and *oo7*, the success rate on detecting the gadgets in our 1.1 million sample data set could serve as a generic evaluation metric while identifying the flaws in the detection tools.

```

1 victim_function:
2     movl    size(%rip), %eax
3     cmpq    %rax, %rdi
4     jae     .B1.2
5     movzbl array1(%rdi), %eax
6     shlq    $9, %rax
7     xorb    %al, %al
8     movb    array2(%rax), %dl
9     andb    %dl, temp(%rip)
10 .B1.2:
11     ret

```

Listing 4.5: *xorb %al, %al* is added to 1st example in Kocher’s examples [104]. *Spectector* is no longer able to detect the leakage due to the zeroing %al register.

4.4 FastSpec: Fast Gadget Detection Using BERT

In an assembly function representation model, the main challenge is to obtain the representation vectors, namely embedding vectors, for each token in a function. Since the skip-gram and RNN-based training models are surpassed by the attention-only Transformer models [220] in sentence classification tasks, we introduce FastSpec, which applies a lightweight BERT version. Transformer models outperform skip-gram and RNN models primarily due to their self-attention mechanism, which captures long-range dependencies and contextual relationships across the entire input sequence, something traditional models struggle with. Unlike RNNs, Transformers process input tokens in parallel, enabling faster and more efficient training on large datasets. Their attention mechanism assigns varying importance to different words, allowing transformers to understand subtle contextual nuances that skip-gram and

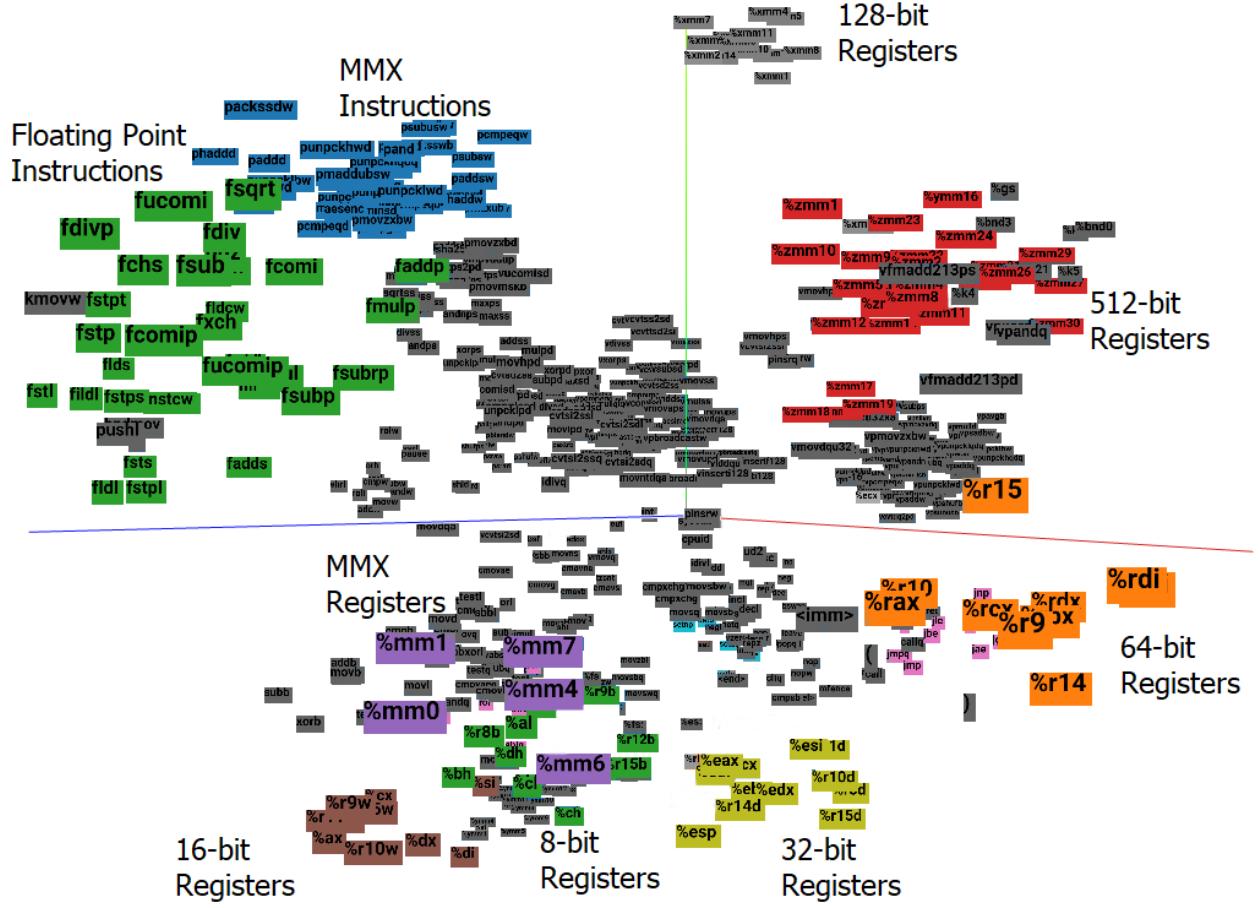


Figure 4.4: 3-D visualization for the distribution of instructions and registers after t-SNE is applied to embedding vectors. Similar instructions and registers have the same colors. The unrelated instructions are separated from each other in the three-dimensional space after the pre-training.

RNN models often overlook. Additionally, Transformers use positional embeddings to capture word order without relying on sequential processing, avoiding the vanishing gradient issues common in RNNs. This architectural simplicity and scalability to large datasets and parameter sizes make transformers significantly more flexible and effective.

4.4.1 Training Procedures

We adopt the same training procedures with BERT on assembly functions, namely, *pre-training* and *fine-tuning*.

4.4.1.1 Pre-training

The first procedure is *pre-training*, which includes two unsupervised tasks. The first task follows a similar approach to MaskGAN by masking a portion of tokens in an assembly function. The mask positions are selected from 15% of the training sequence, and the selected positions are masked and replaced with <MASK> token with 0.80 probability, replaced with a random token with 0.10 probability, or kept as the same token with 0.10 probability. While the masked tokens are predicted based on other tokens' context, the context vectors are obtained by the multi-head self-attention mechanism.

The second task is the next sentence prediction, where the previous sentence is given as input. Since our assembly code data has no paragraph structure where the separate long sequences follow each other, each assembly function is split into pieces with a maximum token size of 50. For the next sentence prediction task, we add <CLS> to each piece. For each piece of function, the following piece is given with the label **IsNext**, and a random piece of function is given with label **NotNext**. FastSpec is trained with the self-supervised approach.

At the end of the *pre-training* procedure, each token is represented by an embedding vector with a size of H . Since it is impossible to visualize the high dimensional embedding vectors, we leverage the t-SNE algorithm [130] which maps the embedding vectors to a three-dimensional space as shown in Figure 4.4. We illustrate that the embedding vectors for similar tokens are close to each other in three-dimensional space, as this outcome shows that the embedding vectors are learned efficiently. In Figure 4.4, the registers with different sizes, floating-point instructions, control flow instructions, shift/rotate instructions, set instructions, and MMX instructions/registers are accumulated in separate clusters. The separation among different types of tokens enables achieving a higher success rate in the Spectre gadget detection phase.

4.4.1.2 Fine-tuning

The second procedure is called *fine-tuning*, which corresponds to a supervised sequence classification in FastSpec. This phase enables FastSpec to learn the conceptual differences between Spectre gadgets and general-purpose functions through labeled pieces. The pieces created for the pre-training phase are merged into a single sequence with a maximum of 250 tokens. The disassembled object files, which have more than 250 tokens, split into separate sequences. Each sequence is represented by a single `<CLS>` token at the beginning. The benign files are labeled with 0, and the gadget samples are labeled with 1 for the supervised classification. Then, the embedding vectors of the corresponding `<CLS>` token and position embedding vectors for the first position are summed up. Finally, the resulting vector is fed into the softmax layer, which is fine-tuned with supervised training. The output probabilities of the softmax layer are the predictions on the assembly code sequence.

4.4.2 Training Details and Evaluation

We combine the assembly data set generated in Section 4.3 and the disassembled Linux libraries to train FastSpec. Although Linux libraries may contain Spectre-V1 gadgets, we assume that the total number of hidden Spectre gadgets is negligible, comparing the data set’s total size. Therefore, the model treats those gadgets as noise, which does not affect the performance of FastSpec. In total, a data set of 107 million lines of assembly code is collected, which consists of 370 million tokens after the pre-processing. We separate 80% of the data set for training and validation, and the remaining 20% is used for FastSpec evaluation. While the same pre-processing phase in Section 4.3.2.3 is implemented, we further merge similar tokens to decrease the total vocabulary size. We replace all labels, immediate values and out-of-vocabulary tokens with `<label>`, `<imm>` and `<UNK>`, respectively. After the pre-processing, the vocabulary size is reduced to 960.

We choose the number of Transformer blocks as $L = 3$, the hidden size as $H = 64$, and the number of self-attention heads as $A = 2$. We train FastSpec on NVIDIA Titan XP

GPU. The *pre-training* phase takes approximately 6 hours, with a sequence length of 50. We further train the positional embeddings for 1 hour with a sequence length of 250. The fine-tuning takes only 20 minutes on the pre-trained model to classify all types of samples in the test data set correctly. Note that the training time is less than previous NLP techniques in the literature since BERT [51] leverages GPU parallelization significantly. The analysis duration is measured on Intel Xeon CPU E5-2637 v2 @3.50GHz.

In the evaluation of FastSpec, we obtained 1.3 million true positives and 110 false positives (99.9% precision rate) in the test data set, demonstrating the high performance of FastSpec. We assume that the false positives are Spectre-like gadgets in Linux libraries, which need to be explored deeply in future work. Moreover, we only have 55 false negatives (99.9% recall rate), which yield a 0.99 F-1 score on the test data set.

In the next section, we show that FastSpec achieves high performance and extremely fast gadget detection without needing any GPU acceleration since FastSpec is built on a lightweight BERT implementation.

4.4.3 Case Study: OpenSSL Analysis

We analyze FastSpec to validate with a separate ground truth data set other than the one we generate in Section 4.3. The purpose of this analysis is to measure the effect of the covariate shift and robustness of FastSpec against a real-world benchmark. We focus on OpenSSL v3.0.0 libraries [168], as it is a popular general-purpose cryptography library in commercial software. We use a subset of functions from RSA, ECDSA, and DSA ciphers in the OpenSSL *speed* benchmark. The function labels are obtained by running the *SpecFuzz* tool, which is a dynamic detection tool to find Spectre-V1 vulnerabilities using fuzzing [159]. The functions in which the *SpecFuzz* tool finds vulnerabilities are labeled as positive, and the remaining ones are labeled as negative. We also exclude the functions without any conditional branch instructions from the positive class and the functions that have a call to them. In total, 4242 functions are extracted from the aforementioned cryptography libraries

to analyze with FastSpec. Positive and negative classes include 720 and 2500 functions, respectively.

First, we apply the same pre-processing procedures, as explained in Section 4.4.2 to obtain the tokens. The total number of tokens is more than 4 million. Since the labels are assigned on function-level, we choose the maximum confidence rate that we get among all the sliding windows. The maximum confidence rate is assigned as the prediction of our model for the corresponding input function. In order to find the optimal sliding window size, we scan through the functions with various different window sizes and compare the performances. Figure 4.5 shows that FastSpec achieves the highest performance to detect functions with Spectre-V1 vulnerability when the window size is set to 80 tokens, which corresponds to 0.998 as an area under the curve (AUC) value. The optimal threshold value is found as 0.48, which corresponds to the maximum F-score. The highest F-score is achieved as 0.99, where the false positive rate (benign functions that are mistakenly classified as Spectre gadget) is 0.04%, and false negative rate (functions that are mistakenly classified as benign) is 2%. We claim that further analysis of the detected functions by using symbolic execution or taint analysis tools can reduce the number of false negative samples and provide an efficient end-to-end security solution against Spectre-V1 vulnerability.

4.4.4 Case Study: Phoronix Test Suite Analysis

The performance comparison between FastSpec and other static analysis tools is evaluated on the Phoronix Test Suite v5.2.1 [136]. For the ground truth, the *SpecFuzz* technique is chosen as the tool that dynamically analyzes the binaries, and exploitable gadgets can be detected with a higher success rate compared to static tools. The selected benign files have source code since it is required to obtain the assembly files for the *Spectector* tool. The assembly files are generated by compiling the source C code with the *GCC* compiler. On the other hand, the binary files are generated at the test installation; therefore, there is no further processing required before testing the binary files in *o07*. For FastSpec, the disassembled

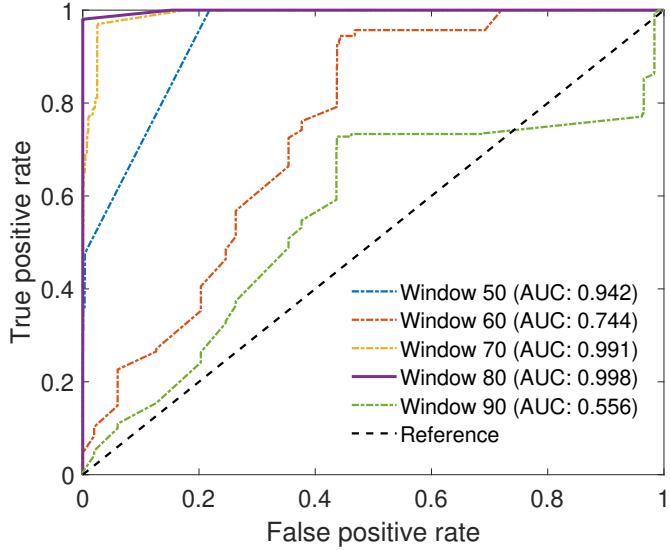


Figure 4.5: Solid line stands for the ROC curve of FastSpec for Spectre gadget class.
Dashed line represents the reference line.

binary files are given as input. Note that since the larger benchmarks take more time to be analyzed by *oo7*, we preferred small size files to make the comparison with *Spectector* and FastSpec easier.

Timing The overall timing results for various benchmarks are given in Table 4.2. The analysis time of *oo7* and *Spectector* increases drastically with the number of conditional branches since the tools analyze both paths after a conditional branch is encountered. On the other hand, FastSpec analysis time increases linearly with the binary size. We observe that the pre-processing phase takes the major portion in the analysis time of FastSpec while the inference time is in the order of microseconds. We fuzz the Crafty benchmark for 24 hours and other benchmarks for 1 hour using *SpecFuzz* under the default configuration ³.

The effect of the increasing number of branches on time consumption is clear in the Crafty and Clomp benchmarks in Table 4.2. Even though the Crafty benchmark has only 10,796 branches, *oo7* and *Spectector* analyze the file in more than **10 days** (the analysis process is terminated after 10 days) and **2 days**, respectively. In Figure 4.6, we show that both tools are not sufficiently scalable to be used in real-world applications, especially when

³<https://github.com/OleksiiOleksenko/SpecFuzz>

the files contain thousands of conditional branches. Especially *oo7* shows an exponential behavior because of the forced execution approach, which executes every possible path of the conditional branches. In contrast, FastSpec analyzes the same Crafty benchmark under 6 minutes, which is a significant improvement.

Note that the Byte benchmark has a higher number of branches than most of the remaining benchmarks. However, it consists of multiple independent files that need to be tested separately, taking less time to analyze in total. Consequently, FastSpec is faster than *oo7* and *Spectector* 455 times and 75 times on average, respectively.

Table 4.2: Comparison of *oo7* [224], Spectector [72], and FastSpec on the Phoronix Test Suite. The last column shows that FastSpec is on average 455 times faster than *oo7* and 75 times faster than *Spectector*. (#CB: Number of conditional branches, #Fc: Number of functions, #DFc: Number of detected functions)

Benchmark	Size (KB)	#CB	#Fc	SpecFuzz		oo7		Spectector		FastSpec	
				#DFc	Precision	Recall	Time (sec)	Precision	Recall	Time (sec)	Precision
Byte	183.5	363	83	7	0.70	0.90	400	1.00	0.43	115	1.00
Clomp	79.4	1464	45	1	0	0	17.5 hr	0.05	0.9	2.8 hr	1.00
Crafty	594.8	10796	207	44	1.00	0.54	>10 day	0.60	0.91	48 hr	0.23
C-ray	27.2	139	11	1	1.00	1.00	395	0.2	0.9	153	0.50
Ebizzy	18.5	104	6	3	0	0	467	0.60	1.00	206	1.00
Mbw	13.2	70	5	1	0	0	145	0.50	1.00	34	0.33
M-queens	13.4	51	4	1	1.00	1.00	136	0.50	1.00	24	1.00
Postmark	38.0	309	49	6	1.00	0.83	3409	0.43	0.95	1202	1.00
Stream	22.0	113	4	3	0	0	231	0	0	63	1.00
Tiobench	36.1	169	19	1	0	0	813	0.25	0.8	201	0.33
Tscp	40.8	651	38	13	0	0	6667	1.00	0.15	972	1.00
Xsbench	27.9	153	32	1	1.00	1.00	1985	0	0	249	0.50
Average					0.47	0.44		0.43	0.67		0.74
											0.87

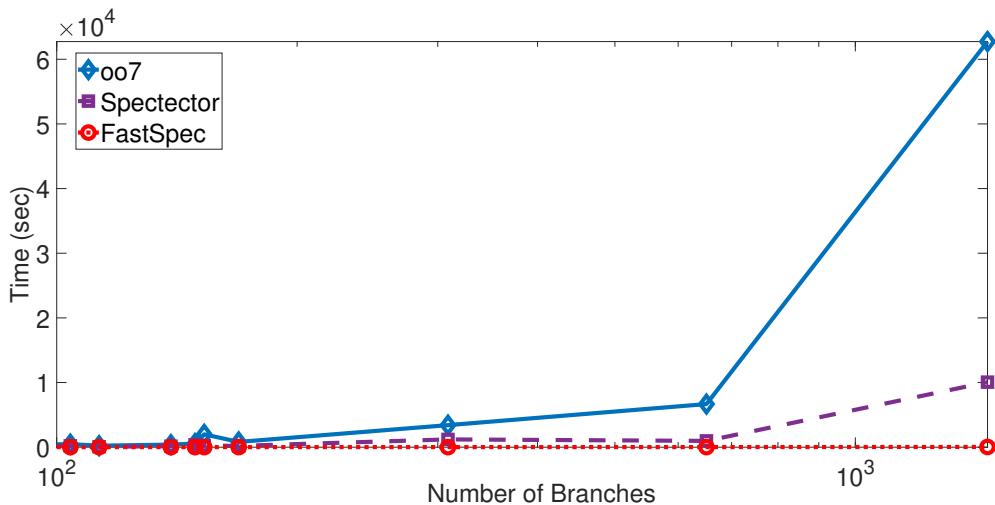


Figure 4.6: The processing time of FastSpec is independent of the number of branches whereas for Spectector and oo7 the analysis time increases drastically.

Baseline Evaluation The number of gadgets found by the tools varies significantly.

While *oo7* and FastSpec report each Spectre gadget in a binary file, *Spectector* outputs whether a function contains a Spectre gadget or not. To be consistent, if a control or data leakage is found in a function, it is reported as a vulnerable function by all three tools.

The precision and recall rates for *oo7*, *Spectector* and FastSpec are given in Table 4.2. The precision is calculated as $TP/(TP + FP)$. TP is the number of overlapping gadgets detected by a tool. FP is the number of functions that are classified as Spectre gadgets mistakenly. The recall value is computed as $TP/(TP + FN)$ where FN is the number of gadgets that are not detected by a tool.

In some cases, *oo7* is not able to track the control flow when the number of function calls increases in a gadget, which yields high false negatives and low recall. Thus, *oo7* suffers from the extraction of complete control flow graph. *Spectector* tends to give more false positives compared to *oo7* and FastSpec. This is because some unsupported instructions are skipped by the tool and the broken Spectre gadgets by specific instructions are still classified as Spectre gadget. On the other hand, FastSpec has low false negatives since all the Spectre gadget patterns are detected with a confidence rate higher than 0.48. When the file size increases, the false positives may increase in parallel. However, these gadgets can be verified with other tools to increase the confidence. As a result, FastSpec scans the functions

extremely quicker than other tools without sacrificing the precision and recall rates. Our tool also guarantees the security of the scanned assembly functions by detecting almost all Spectre gadgets with low FN rates. FastSpec outperforms all the compared tools in terms of recall and precision rates by a large margin.

4.5 Discussion and Limitations

4.5.1 Gadget Verification

The gadget verification process in Section 4.3.1 is implemented in an isolated core since the system interrupts frequently mistrain the targeted branch instructions in the gadgets, which decreases the gadget verification success rate significantly. This situation particularly affects the first step of the verification process where all the inputs are out-of-bounds, and the target branch is not expected to be mistrained. Therefore, there is a need for an isolated environment to run the verification code for Spectre gadgets. Even though the data cache side-channel is used for the secret decoding, other side-channels can be used to decode the secret in a Spectre gadget such as TLB structure. The secret elements in *array1* should be multiplied with a constant to decode the secret into different cache lines or pages. In the base examples [104], the secret elements are multiplied by 512 or 4096. The verification code only selects the Spectre gadgets with these specific multiplicands, which potentially introduces a bias in the data set. Since all multipliers in the Spectre gadgets are represented with the same token, `<imm>`, our detection tool is not affected by the bias introduced by different multipliers. For instance, in OpenSSL and in Phoronix, we observed that gadgets with different multiplicands are detected by our detection tool.

Our verification codes also focus on more complex leakage snippets in which the secret is not simply leaked with a simple multiplication. We observed that similar control-flow statements and more complex encoding techniques are present among Kocher’s examples [104] (Examples 10–15). After new gadgets are generated from these examples, we observed that

these gadgets can still be detected by our verification code. However, if the leakage mechanism in the gadget is altered significantly, it is likely that the secret in the generated gadget is not recovered during verification. Unfortunately, this introduces a bias in our data set as the diversity of the gadgets is limited. Moreover, our detection tool might not be able to detect more complex gadgets as these gadgets are not included in the training data set. To include more complex gadgets in the data set, the verification code can be changed dynamically by analyzing each generated assembly code, which is left as future work.

4.5.2 Scalability and Flexibility

Other Spectre Variants: Since pre-training teaches the general assembly syntax and takes a major part in the training process, our pre-trained FastSpec model can be used after fine-tuning for any assembly code task. The modifications are needed only to Step 1 and Step 4 in Section 4.3.1 since we need an initial data set and verification code to build up a larger data set. For Spectre v1.1 [102], our verification code can be adapted by adding one more attacker-controlled input to verify whether a speculative load is executed or not. Similarly, the speculatively written value in Spectre v1.2 [102] can be mapped to cache lines to verify the generated gadgets. For Spectre v2 [105], verification procedure needs to be completely changed as the branch instruction is not a conditional branch anymore. For this purpose, the verification code can be modified to mistrain the indirect jumps with attacker known addresses, and then, the secret bytes in the attacker-controlled function are mapped to separate cache lines. Since Spectre-RSB [107] works in a similar way, except *ret* instruction is targeted, the same verification procedure can be adapted. Finally, in Spectre v4 [82], the verification code can supply attacker-controlled variables to specific registers, and then, speculatively loaded data can be decoded to a shared memory to verify the gadgets.

Other Attacks: Our approach can detect the target **SMoTher**-gadgets [22] in the code space. The verification procedure in Section 4.3.1, specifically Step 4, needs to be changed to analyze port fingerprints. For this purpose, the timing of various instructions that are

mapped to certain ports can be measured to detect the leaked secrets as implemented in [11]. It is highly likely that the verification takes more time for the generated gadgets since we need to collect more timings to distinguish the cases between secret leakage and no secret leakage. In NetSpectre [189], there are two types of gadgets. The leak gadget is very similar to Spectre v1 whereas only one bit is transmitted. Hence, the verification procedure can be modified to profile a single cache line instead of 256 cache lines. The transmit gadget is used to leak the secret data over the network and has a different structure than the leak gadget. To detect the transmit gadgets with our verification code, the Thrash+Reload technique can be adapted to measure the timing difference between cached and non-cached accesses over the network. Again, the verification procedure potentially takes more time to analyze the generated gadgets since the secret transmission speed is significantly lower than Spectre V1.

Other Architectures and Applications: Although we limit the scope of this work to generating and detecting the Spectre-V1 gadgets on x86 assembly code, the use of SpectreGAN and FastSpec can always be extended to other architectures and applications with only mild effort. Furthermore, specially designed architectures are not needed when pre-trained embedding representations are used [51]. Therefore, the pre-trained FastSpec model can be used for any other vulnerability detection, cross-architecture code migration, binary clone detection, and many other assembly-level tasks.

The fuzzing tool increases the diversity of the generated gadgets by introducing variations that are later learned by the FastSpec tool. In addition, the detection tool learns the generic gadget type rather than training on small details. In Section 4.4.2, the evaluation of FastSpec also shows that the tool can detect the potential Spectre gadgets with a 99.9% precision rate.

4.5.3 Comparison of FastSpec with Other Tools

The most significant advantage of FastSpec is the capability of detecting Spectre gadgets quicker than other tools. If an instruction is not introduced in the training phase, the instruction is treated as unknown, and it has a slight effect on the accuracy of FastSpec

since a large window of instructions is analyzed to decide on the Spectre gadgets. While the unsupported instructions are an important issue for the *Spectector* tool, FastSpec can be deployed to other architectures such as ARM and AMD. While small modifications in the assembly code increase the chance of bypassing other tools, our tool is more robust against small modifications. It is easier to adapt FastSpec to other Spectre variants as the vector representations of assembly instructions can be directly used to train a separate model for the variants. Moreover, over-tainting and under-tainting issues decrease the accuracy of taint-based static analysis techniques. However, FastSpec tracks the registers, instructions, and memory accesses with a vector representation, which makes it more reliable in large-scale projects.

4.5.4 Scope and Limitations

Scope: Our scope is to generate Spectre-V1 gadgets by using mutational fuzzing and SpectreGAN methods as well as to detect potential Spectre gadgets in benign programs by significantly reducing the analysis time.

Guarantees: Our verification methods in Step 4.1 guarantee that the generated Spectre-V1 gadgets leak the secret bytes through cache side-channel attacks. Moreover, the FastSpec tool detects the Spectre gadgets with a high precision and recall rate by identifying the gadget patterns at the assembly level. Possible False Positive outputs do not affect the security guarantee provided by FastSpec. The analysis time is significantly reduced compared to rule-based detection tools.

FastSpec generalizes well, i.e., it can recognize similar patterns that are not in our training dataset. However, it does not provide assurance of coverage (completeness) since FastSpec is not based on hand-written rules or formal analysis. In order to decrease the False Negative rate, the probabilistic threshold is kept low in the case studies. In contrast, while FastSpec does not provide such guarantees, it is much faster and scales to larger code-bases.

Assembly Code Generation: VAEs are widely used for learning data representations

and have been applied to text and image generation tasks. However, they have some key limitations. For example, VAEs often smooth out details in their outputs, which can make them less suitable for applications that require fine-grained precision. They also face an issue known as posterior collapse, where the latent space contributes little to the final output, reducing their effectiveness for tasks that demand high levels of accuracy and detail. This also makes it less suitable for increasing the diversity of the generated gadgets.

The challenges faced in the regular text generation with GANs [56, 246] also exist in assembly code generation. One of the challenges is *mode collapse* in the generator models. Although training the model and generating the gadgets with masking help reduce mode collapse, we observed that our generator model still generates some tokens or patterns of tokens repetitively, reducing the quality of the generated samples and compilation and real gadget generation rates.

In regular text generation, even if the position of a token changes in a sequence, the meaning of the sequence may change while it would still be somewhat acceptable. However, if the position of a token in an assembly function changes, it may result in a compilation error because of the incorrect syntax. Even if the generated assembly function has the correct assembly syntax, the function behavior may be completely different from the expected one due to the position of a few instructions and registers.

The fuzzing-based gadget generation technique is based on known gadget examples. Since there are already 15 versions of Spectre-V1, we use these gadgets as the starting point for fuzzing. On the other hand, the available gadgets for other variants are significantly lower compared to Spectre-V1 gadgets. To solve this issue, other detection tools can be used to detect Spectre gadgets in benign programs. Then, new gadgets can be generated with fuzzing technique. We leave the further investigation of generation other Spectre variants as future work.

Recently, decoder-only Transformer models, such as GPT-4, are shown to be superior to GANs in terms of text and code generation tasks. The use of more advanced language

models can help creating a more diverse and realistic data set and, ultimately, it can make the classifier model more performant. Exploring the use of Transformer models for generating Spectre gadgets is left as a direction for future research.

Window Size: Since Transformer architecture has no utilization of recurrent modeling as RNNs do, the maximum sequence length is needed to be set before the training procedures. Therefore, the sliding window size can be set to at most the maximum sequence length. On the other hand, our experiments show that using lower window sizes than maximum sequence length provides more accurate Spectre gadget detection and provides fine-grain information on the sequence.

4.6 Conclusion

This work, for the first time, proposed NLP inspired approaches for Spectre gadget generation and detection. First, we extended our gadget corpus to 1.1 million samples with a mutational fuzzing technique. We introduced the SpectreGAN tool that achieves a high success rate in creating new Spectre gadgets by automatically learning the structure of gadgets in assembly language. SpectreGAN overcomes the difficulties of training a large assembly language model, an entirely different domain than natural language. We demonstrate 72% of the compiled code snippets behave as a Spectre gadget, a massive improvement over fuzzing based generation. Furthermore, we show that our generated gadgets span the speculative domain by introducing new instructions and their perturbations, yielding diverse and novel gadgets. The most exciting gadgets are also introduced as new examples of Spectre-V1 gadgets. Finally, we propose FastSpec, based on BERT-style neural embedding, to detect the hidden Spectre gadgets. We demonstrate that for large binary files, FastSpec is 2 to 3 orders of magnitude faster than *oo7* and *Spectector* while it still detects more gadgets. We also demonstrate the scalability of FastSpec on OpenSSL libraries to detect potential gadgets.

Chapter 5

Automated Side-Channel Patching in Source Code Using LLMs

5.1 Motivation

The advent of microarchitectural attacks has instigated efforts to mitigate vulnerabilities in hardware/firmware and in deployed software libraries. Earlier vulnerabilities, such as those exploiting secret dependent execution time and cache/memory access patterns, were followed by more advanced attacks exploiting microarchitectural optimizations such as out-of-order and speculative execution [105, 119], transient write forwarding and shared buffers [27, 187, 219].

One of the earliest and still most accessible forms of side-channel leakage is execution time. If a developer inadvertently writes code, e.g., with secret data-dependent branches, by measuring the execution time, an attacker can deduce secret information. Therefore, identifying vulnerable software and replacing them with their constant-time versions has been a goal of security researchers. This is challenging in practice since repositories have complex interdependence with many potentially vulnerable pieces, while their execution time is also dependent on many factors, e.g., the platform and its configuration, the compiler.

Spectre was first discovered and publicly disclosed by security researchers in the original Spectre paper in 2018 [105]. Spectre v1 occurs when attackers can trick the CPU into speculatively executing code that would not normally be run during normal program execution. By exploiting this vulnerability, attackers can potentially access sensitive data or information stored in the memory of other applications or the operating system. The attack leverages the processor’s speculative execution to infer and exfiltrate this sensitive data.

In his blog, Kocher [104] shared 15 code snippets vulnerable to variations of Spectre v1 (Spectre gadgets) to test out a new version of Microsoft VC/C++ compiler with built-in mitigation [165] based on the addition of the LFENCE instruction to sensitive parts of the code identified by Microsoft’s static analyzer. The compiler only manages to mitigate Spectre in the first two gadgets. Kocher points out that his code samples are far from comprehensive, e.g., they all rely on cache modification as a covert channel, and they all reside in simple functions more amenable to static analysis. Cauligi et al. [32] present a comprehensive survey of existing Spectre v1 defenses and non-constant time detection tools e.g. oo7 [225], Spectector [72], SpecFuzz [160], Pitchfork [30].

Code with microarchitectural vulnerabilities, e.g., secret dependent non-constant time or code vulnerable to Spectre v1 has since been a significant concern for the tech industry. Hardware and software vendors have released mitigations to reduce the risk of exploitation, but fully addressing these vulnerabilities remains an ongoing challenge. Moreover, these mitigations often come at the cost of decreased performance, as they may disable or limit certain speculative execution features.

In a study among the crypto library developers, 61.4% of the participants stated that they are either not aware or they do not use the tools for testing and verifying the constant-timeness [93] – a necessary but insufficient condition for side-channel security. To make matters worse, many of these libraries that are used by millions of end-users are managed by a small number of developers in open-source projects. They neither possess the knowledge nor the resources to patch their software against such low-level leakages. Often times reported

vulnerabilities go ignored and unpatched in publicly available open-source crypto libraries used by millions, e.g., see Microwalk-CI [232], due to lack of resources. Another striking example is in the OpenSSL Blog Post [44] explaining their decision on why they chose *not* to patch for newly discovered Spectre gadgets reported in [146] : “*Most potentially vulnerable code is extremely non-obvious, even to experienced security programmers. It would thus be quite easy to introduce new attack vectors or fix existing ones unknowingly.*” and “*Automated verification and testing of the attacks is necessary but not sufficient. We do not have automated detection for this family of vulnerabilities, and if we did, it is likely that variations would escape detection.*”. These comments highlight the need for reliable and transparent patch automation.

In this work, we study the use of LLMs for automated patching of security-critical software. Indeed, it is expected that 80% of the software development lifecycle will use generative AI, i.e., LLMs, by 2025 [62]. Thus, evaluating LLMs’ capability to generate security-critical implementations is an urgent need. What happens if we use ordinary prompts to generate crypto code, and how can we improve code generation to improve side-channel security while ensuring functional correctness? We are encouraged by rapid advances in LLMs. Fueled by recent innovations in Transformer networks, generative models, and the availability of massive datasets and large compute clusters, it has become possible to train Large Language Models (LLMs). LLMs such as GPT3 [26] and GPT4 [161] by OpenAI, BERT [51] and PaLM2 [12] by Google, RoBERTa [127] and LLaMA [210, 211] by Meta AI have shown impressive performance in AI applications and in natural language processing (NLP). These tools are also trained using code snippets, allowing one to parse and even generate code in common programming languages flexibly.

In this work, we study the use of LLMs in concert with state-of-the-art leakage and vulnerability detection tools to fix data-dependent non-constant time behavior, as well as secret-dependent branching and Spectre v1 gadgets. Such vulnerabilities are known to exist in numerous security libraries deployed on millions of machines. Yet, due to the lack of

resources, i.e., experts and financial resources, they go unpatched. Our goal is to make use of the massive recent advances in LLMs such as OpenAI GPT, Google PaLM, and Meta LLaMA to generate patches automatically. Note that LLMs are fairly large, and it takes weeks to months to train on massive datasets, resources that only large companies have access to. Our goal is to utilize LLMs via API access to bring down the cost of patch deployment to cents per microarchitectural leakage.

Contributions

- We present the first comprehensive study of state-of-the-art LLMs, i.e., OpenAI GPT, Google PaLM 2, and Meta LLaMA, to automatically patch microarchitectural vulnerabilities such as secret dependent (non-constant time) code and Spectre v1 gadgets.
- We build a toolchain that tests binaries for leakage and Spectre detection tools, specifically Microwalk [232], Pitchfork [30], Spectector [72], and KLEESpectre [223], and then automatically generates security patches to be included in the source files using LLMs.
- From a Continuous Integration/Continuous Development (CI/CD) perspective in the software development life cycle, the proposed framework allows us to patch the source code (e.g., C/C++, Javascript, etc.) while testing the binary after compilation on a target machine. Compared to binary patching, we retain the ability to review and revise the source. At the same time, we are also taking into account the effect of the compiler and platform configuration on security and efficiency by testing the binary for leakage. This approach allows us to continuously improve the software as hardware systems and software stacks evolve.
- On a microbenchmark of C code we compiled from known vulnerabilities, GPT4 excels in patching 97% of all leakages successfully of every type of patching points in the benchmark, while the total cost of patching 33 leaks is at \$1.34. GPT3.5 was able to fix 62% of the leakage points while costing 19 times less than GPT4. Google `chat-bison` and Meta LLaMa2 patch 56% and 35% across all vulnerabilities, respectively, albeit at much lower

cost.

- Our framework is only limited by the capability of the detection tools, e.g., false positives and negatives, and will rapidly improve further with better detection tools. Similarly, LLMs are improving at an astounding rate (almost every month, a new LLM is released), and we expect significant improvement in the overall performance of our tool.
- From an efficiency perspective, with up to $\sim 10\times$ faster than Spectre v1 patches generated with existing methods, our toolchain significantly outperforms compiler-based techniques such as in `clang lfence` injection. Hence, the proposed approach provides an opportunity to remove unnecessary inefficiencies while retaining security.
- Since we are patching the source code with the output generative LLM, the patches are also commented, which makes it easier to make sense of the patch and maintain the code.

5.2 Related Works

The field of automated program repair has seen various advances, but these studies typically focus on syntactic and build errors, with fewer exploring the domain of security vulnerabilities, and none, to date, have addressed the issue of microarchitectural vulnerabilities.

DeepFix, as Gupta et al. [77] proposed, aims to automatically correct common programming errors using a sequence-to-sequence neural network model. However, this method is fundamentally limited in scope, as it does not tackle any security vulnerabilities. Its performance is also contingent on the accuracy of error location prediction, which is inherently challenging. Similarly, the Break-It-Fix-It (BIFI) method by Yasunaga et al. [242] primarily targets syntactic errors, leaving the important domain of security vulnerabilities unaddressed. Moreover, despite improving over previous methods, BIFI’s repair accuracy still leaves a significant percentage of errors uncorrected, pointing towards a potential need for better training methods and error diversity. The Graph2Diff model introduced by Tarlow et al. [200] extends the focus to build errors but continues to overlook security vulnerabilities.

The model’s effectiveness is also potentially limited in complex scenarios, where precise diff prediction might not be sufficient or even feasible.

The study by Pearce et al. [166] is particularly noteworthy as it forayed into the realm of security vulnerabilities. Their use of LLMs for zero-shot vulnerability repair is indeed promising. However, their focus is largely limited to basic software bugs, which, while important, is only a subset of the challenges developers face. Despite the potential demonstrated by LLMs, the study did not extend their use to more complex and critical issues, such as microarchitectural vulnerabilities and sophisticated crypto implementations.

Coming from the hardware perspective, Ahmad et al. [10] consider how LLMs may be leveraged to repair security-relevant bugs present in Verilog models automatically. In particular, they explore the prompt space to show that by using OpenAI’s Codex, one may outperform the Cirfix hardware bug repair tool on its own suite of bugs. For Java code repair, Wu et al. [238] analyze five LLMs and existing automatic program repair (APR) tools on two real-world benchmark tools. They find that out of the box, both LLMs and APR fix only a small fraction of vulnerabilities (about 20% for Codex) but also note that fine-tuning LLMs using APRs does improve the performance. The study by Charalambous et al. [35] investigates us of LLMs, specifically GPT3.5-turbo, and formal verification checkers, i.e., Efficient SMT-based Context-Bounded Model Checker (ESBMC), to fix vulnerabilities in C. The proposed method achieves an impressive success rate of up to 80% in repairing vulnerable code with buffer overflow and pointer dereference failures.

Garg et al. [61] focus on fixing hard-to-detect performance bugs in C# software with zero-shot LLMs. They take a slightly different approach: given a line of code that contains a performance bug, the line is compared to lines in a pre-constructed knowledge base to retrieve a prompt command that can be used to convey what change needs to be fed into an LLM. Using OpenAI’s Codex, their tool can generate performance improvement suggestions equivalent to or better than a developer in 60% of the cases.

Kande et al. [96] study the use of LLMs for the automatic generation of hardware as-

sertions (in SystemVerilog) for vulnerability testing of production-grade hardware. Their proof of concept study uses OpenAI’s Codex `code-davinci-002` LLM, generating 75,600 assertions and generating correct assertions 4.53% of the time. They note that while the assertion rate is small, further optimization can improve the rate.

Despite substantial advances in automatic program repair, a clear gap persists in addressing complex security and especially microarchitectural vulnerabilities in intricate cryptographic implementations. While LLMs show promise, their capabilities need to be further explored and expanded to tackle these complex and critical challenges effectively. This forms a compelling motivation for our work.

5.3 Threat Model and Scope

In this work, we focus on preventing secrets from being leaked through the changes observable to software. Using microarchitectural side-channels, attackers can obtain sensitive information such as encryption keys, passwords, etc. We assume that the attacker wants to exploit a certain side channel on the system, and the attack requires security-critical software that exhibits one or more of the following properties,

- Code access patterns depend on the secret,
- Data access patterns depend on the secret,
- The execution time of the code depends on the secret.

Although it is possible that even if none of these properties exist in logical channels, the underlying hardware implementation can cause physically visible leakages, such as through power and electromagnetic emanation, we only consider software-enabled leakages in this work.

We also assume that the software is free of bugs and works in the intended way. Therefore, common software bugs, such as buffer overflow, use-after-free, etc, are not considered in this work. We assume that the attacker has the capability to measure the execution time of the

software or collect other kinds of metadata through shared system components such as CPU cache and deduce sensitive information through secret data-dependent branches, memory access patterns, or by exploiting speculative execution.

We explore the use of state-of-the-art LLMs to improve the resiliency of security-critical software against these microarchitectural attacks. Since training LLMs from scratch is costly, time, and energy-consuming and bears an environmental impact [211], we leave custom-trained LLMs out of scope and focus on only prompting. Note that the models we have evaluated are not tuned for patching security vulnerabilities, yet their pre-training dataset is likely to contain documentation and code bases that are related to security.

5.3.1 Research Questions

In this scope, we focus on the following questions:

Q1 Using LLMs, can we gain the ability to patch large-scale software against microarchitectural vulnerabilities?

Q2 How well do LLMs perform for side-channel patching across different programming languages?

Q3 What is the cost of LLM-based patching? How does it compare in reliability, cost, and speed against human experts?

Q4 How does the patching performance vary across LLMs?

5.4 Methodology

5.4.1 Ensuring Constant-Time Execution

Since the emergence of timing side-channel attacks [106], many tools have been proposed to validate the constant-time (data oblivious) property of software. Nevertheless, the burden of implementing constant-time code predominantly rests on software engineers to this day. Consequently, numerous security-critical libraries lack any form of testing within their CI/CD

pipelines for constant-time property [30]. To the best of our knowledge, for the first time, we propose an automated tool that generates constant-time implementation based on LLMs.

5.4.1.1 Evaluating Side-Channel Leakage

We address a side-channel leakage by assuming a robust adversary (evaluator) with extensive access to runtime events, including memory accesses and the execution path. The adversary can also select and modify any secret system input. In the context of cache attacks, the adversary treats memory accesses as a leakage vector, gathering all memory accesses throughout the execution with various secret values. If a relationship between different secrets and memory access variation is found, the adversary can pinpoint instructions related to secret-dependent memory accesses and reveal potential leakages. Various tools exist in the software verification landscape to detect such leakages, each capable of ascertaining the constant-timeness of software [93]. The selection of a specific tool is contingent upon the particular needs and constraints of the task at hand. In our case, we employ *Microwalk* [231] due to its blend of benefits while acknowledging its limitations. *Microwalk* leverages mutual information, a robust measure that allows us to quantitatively assess the extent of information leakage, providing a clear and interpretable metric. Additionally, *Microwalk* can capture a wide range of potential leakages, including those from the execution path and memory accesses. Most importantly for our use case, it can localize the source of leakage in the binary and source code (if available). However, it is worth noting that *Microwalk* requires executing the target binary multiple times to accurately estimate mutual information, which can increase the computational costs. Hence, our choice balances comprehensive leakage detection, quantitative assessment capability, and computational feasibility.

Microwalk first generates arbitrary inputs for a given secret. Following this, the target binary is run on each input collecting data on memory allocations, branches, calls, returns, memory reads and writes, and stack operations in each run. Ideally, constant-time implementations should have a linear execution path for secret input. Secret-dependent conditional

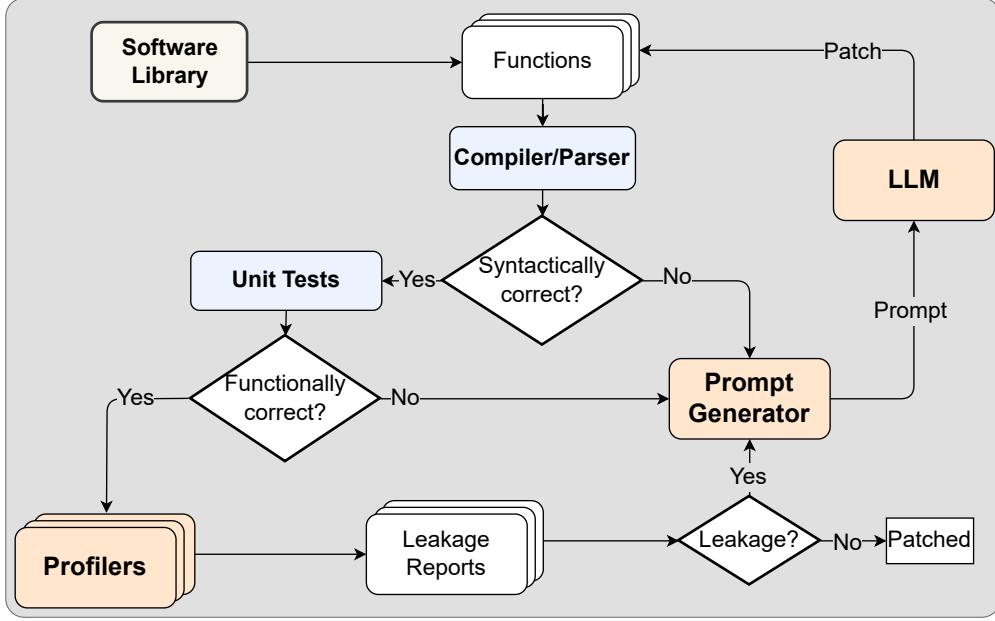


Figure 5.1: ZeroLeak patch generator framework overview.

branches leak information about the secret. By considering the execution path as a leakage vector, we can confirm whether the same operations are performed for any secret input. Another common leakage source, memory access, should follow a secret-independent pattern in constant-time implementations. Hence, we ensure memory accesses are either constant or at least not correlated to the input.

5.4.1.2 Patching for Constant-timeness

Three main challenges need to be addressed for automating the constant-time patches using LLMs.

Challenge C1 First, patching common software bugs in simple programs often can be resolved by changes in a few lines of code, which LLMs were shown to be capable of [166]. However, making a software implementation of an algorithm constant-time is far more complex since it requires a deep understanding of algorithm logic, and keeping track of how and where the secret is used. Also, a single code may have multiple points which contribute to the overall leakage. Therefore, LLMs do *not* perform well in fixing a side-channel leakage in

a complex implementation in a single shot.

Challenge C2 Second, simply stating that the code is showing observable traces that are correlated to the secret is not enough to patch a complex logic. This is also one of the reasons why human developers have difficulty creating a constant-time code without localizing the leakage points. Therefore, it is essential to localize the leakage points in the code for efficient and effective patches for LLMs as well.

Challenge C3 Finally, prompts should be crafted in the proper way that explains the reason for the leakage in the most precise and clear manner without leaving any ambiguity. For example, instructing the LLM to “make the code constant-time” alone in the prompt without giving any security context can cause misinterpretation of constant-timeness in the context of time complexity, i.e., that the run-time complexity of the algorithm should be $O(1)$. This is clearly insufficient since we want the run-time to be independent of the actual input values.

We overcome **C1** by adopting an iterative approach. Since many of the LLMs are designed as a chatbot, they perform better in a conversation with back-and-forth message exchange and with feedback from a human. Since we aim to replace humans in the patching process with a tool, we can run the generated code on the target platform with the analysis tool and get feedback without any cost. We use a patching loop that is illustrated in Figure 5.1 that works as follows:

- Assuming we are testing a function in a library, we first make sure the function is called from within the *Microwalk* template and unit tests are ready to verify the correctness of the code. The analysis template can also be generated using LLMs.
- Then, we compile the code if necessary and run *Microwalk* on it. Assuming the first version is already correct, our tool starts parsing the analysis files and passes the vulnerable functions to LLMs together with prompts so they can generate patched code.

- The patched code is verified if it is syntactically correct using parsers/compilers. If the syntax is wrong, we give feedback to LLM until it generates a syntactically correct code. If the syntactically correct code fails the functional correctness tests embedded in the *Microwalk* template, it is forwarded to LLM again as well.
- The loop ends when there is no vulnerability found, but under limited resources, iteration counts and total execution times can be limited.

We also append the responses given by the LLM when it is syntactically correct. As the loop continues, the context given to LLM looks like [System, User, Response, User, Response, ...], which is a common practice in chatbot applications. If the context size reaches the maximum token count of the model, we start dropping from the third message and forward to keep the system prompt and the original function in the context all the time.

We address **C2** by choosing an analysis tool that is capable of localizing the leakage points in the binary and source code. *Microwalk* is a suitable selection for this purpose. The Javascript version can tell exactly which line in the source causes the leakage. The C version, on the other hand, can mark the leakage source at the assembly level. To translate the assembly lines to C source code, we compile it with debug symbols and disassemble the binary using `objdump`. More advanced reverse engineering tools, such as Ghidra [63] or IDA [80], can also be used for more accurate results. After disassembling, we create a mapping of the assembly lines to C lines for use in prompts later.

For **C3**, we use *Microwalk*'s analysis results, which show the exact leakage points as code lines and categorize the leakage mechanism to certain classes, such as memory access-based and conditional execution. We incorporate the analysis results into natural language, which LLMs can understand better, as shown in Figure 5.2. We give a system prompt to the model but with additional commands that prevent common mistakes. We identified mistakes such as

- generating only the patched portion of the code because the rest is unchanged,
- calling a hypothetical function or variables that are not defined,

- changing the number and types of arguments to the given function, and changing the name of the function,

which all break the program’s compatibility with the rest of the library. We also describe how new functions can be added if required. Without this command, the model can give a new function without integrating it into the main function, which also causes crashes when we directly overwrite the main function. Finally, we include tool and language-specific commands, shown as <specifcs> in Figure 5.2, which are not necessary to generate a secure/functional code but are required to resolve the compatibility issues, e.g., new features like `let`, which was introduced with *ES6* to Javascript causes crashes in *Jalangi2* which *Microwalk* backend is based on for Javascript.

When formulating prompts for patching the side-channel leakage, we consider the following options in the user prompt:

Option 1 – Leaky Memory Access Pattern: After giving the full function, we list the name of arrays in the line of code and give the full line and instruct the model to make the memory accesses independent of the secret.

Option 2 – Leaky conditional executions: For this case, we parse the `if`/ternary from the line and instruct the LLM to implement it without `if` statements and ternary operators.

Option 3 – Secret dependent loop size: We parse the termination condition in the loop and instruct the model to keep the number of iterations fixed for every input.

Option 4 – Syntactically/Functionally incorrect code: Some iterations may generate syntactically incorrect code, which can be detected even without running it. We use the feedback from the parser/compiler for the next iteration’s prompt to avoid losing the attempt to patch other bugs since they might still be logically correct. Some iterations may generate functionally incorrect code, which can be detected during the run time. For that, we use assert statements in the test benches and set the <crash reason> as *The code is not working correctly..*

Since options are limited in this scenario, semi-adaptive prompt crafting based on a

```

1 System Prompt:
2 You are an expert at implementing constant-time
3 cryptographic algorithms in <language>.
4 Patch the given functions according to user's
5 instructions. Do not give detailed explanations.
6 The generated code should be complete, do not omit
7 any part of the code. It should be able to run
8 without any post-processing. You can implement new
9 functions and integrate them with the original
10 function. Do not introduce new arguments to the
11 given function. Do not change the name of the
12 function. <specifics>
13
14 User Prompt:
15 <Option 1>
16 <function to patch> <array names> array is
17 accessed dependent on the secret in line <line>.
18 Patch the code such that the array access is made
19 input independent.
20 <Option 2>
21 <function to patch> The condition in
22 <if statement> is secret dependent and causes
23 side channel vulnerability. Patch the code such
24 that it does not require any conditional execution.
25 <Option 3>
26 <function to patch> The termination condition in
27 <loop statement> is secret dependent. Patch the
28 code such that loops execute the same amount of
29 time for every input.
30 <Option 4>
31 <crash reason> The generated code must be complete.
32 Generate everything even if you do not make any
33 changes. Try the same patch again.

```

Figure 5.2: Prompt template for constant time patch. We replace <language> with the programming language, such as C or Javascript. We use <specifics> for instructing workarounds for the tool or language-specific compatibility issues. Other variables are self-explanatory.

template works well. For a more adaptive system, prompt design can be outsourced from generative AI and by chaining LLMs [236, 237]. Although the prompt templates we propose are based on expert knowledge, the solution is scalable to large code bases since the options provided in the templates cover all possible ways of leakages that the detection tools can find.

5.4.2 Mitigating Spectre-v1

Scalable mitigations to Spectre-v1 come with a cost of high overhead due to too generic design. On the other hand, low-overhead solutions such as index masking require manually changing code. Even after manually adding the mitigation in the source code, the effect of the mitigation on the binary is often overlooked. One such example of the failure of relying on manual fixes on source code without testing on binary was discovered by [67] on the Linux kernel. After the emergence of Spectre attacks, Linux developers added a new API that implements `array_index_nospec` macro to clamp the indexes to the arrays to maximum array size. Although it is a correct fix, in one case, it was found to be eliminated by the compiler because the compiler semantics is not aware of speculative execution, and it can optimize out a critical attack mitigation. Hence, in this section, we will focus on how we can automate low-overhead software mitigations using LLMs that are reliably verified on the binary.

5.4.2.1 Finding Spectre-v1 Gadgets

Finding Spectre-v1 gadgets in a scalable and sound way remains an ongoing research area. However, to automate the patching process for Spectre-v1 gadgets, we need a tool that is both scalable and sound. In this work, we evaluate the usage of several analysis tools, such as Pitchfork [30], Spectector [72], and KLEESpectre [223], which covers different aspects of state-of-the-art detection tools, such as security guarantees, scalability, detection method, out-of-order execution support, handling non-determinism, and leakage model [31].

Although Pitchfork also supports the Spectre STL (Store-to-Load) variant, we only consider PHT (Page History Table), the common variant supported by all three tools. Both Spectre and Pitchfork use a hardware-agnostic constant time leakage model. KLEESpectre detects if data leakage caused by the speculative execution is visible to the attacker by extending symbolic execution with micro-architectural features, i.e., cache, and tests each way of every conditional branch (taken or not taken). It assumes the branch predictor will always mispredict.

5.4.2.2 Patching Spectre-v1 Gadgets

Although discovering Spectre-v1 gadgets presents significant challenges, devising mitigation strategies for these gadgets is equally challenging. In this work, for the first time, we propose using LLMs to patch functions with known leakage points in the transient domain.

Most of the challenges in patching Spectre gadgets overlap with generating constant time crypto implementations that we explained in Section 5.4.1. Therefore, the overall ZeroLeak framework in constant time will apply here as well, with different tools instead of *Microwalk* in Figure 5.1. Since all the tools we analyzed are capable of extracting symbolic execution trees, they can pinpoint leakage sources at the assembly level. From assembly, we use the same approach in 5.4.1.2 to trace it back to the source code.

Our design in prompt template changes according to the speculative leakage mechanism caused by conditional branches. The system prompt we use is very similar, except we replace “constant-time” with “secure” since we do not want to instruct the model that there is a non-speculative leakage in the given code. Note that the leakage mechanism in non-speculative scenarios involves secret inputs given to the program. However, the inputs are controlled by the attacker in Spectre-PHT and are not considered secret. For the user prompts, we consider the following two options that are illustrated in Figure 5.3:

Option 1 – Spectre-v1 Violation: After giving the full function, we parse the statement that includes `if` condition or ternary operators, which are translated as conditional branches

```

1 User Prompt:
2 <Option 1>
3 <function to patch>
4 <conditional statement> can be speculatively
5 executed when the condition inside is wrong. Fix
6 the code such that the condition is checked
7 without an if statement or ternary operator.
8 <Option 2>
9 <crash reason> The generated code must be complete.
10 Generate everything even if you do not make any
11 changes. Try the same patch again.

```

Figure 5.3: Prompt template for patching Spectre-v1 gadgets.

in the binary by the compiler. We mention that speculative execution may cause incorrect executions even if the condition is wrong and instruct the model to replace the conditional statement. Although more detailed prompts that include further details, such as which array is indexed and how it is decoded, may sound more intuitive, we choose a more generic and precise prompt that is less likely to confuse low-capacity models; see Section 5.5.4.

Option 2 – Syntactically/Functionally incorrect code: We use the same approach as in Section 5.4.1.2.

5.5 Evaluation

We evaluate ZeroLeak on both non-constant time code and Spectre gadgets. We design our experiments in incremental hardness.

Experiment Setup For leakage quantification for constant-time code, we have used docker images of Microwalk packages with version 3.1.1-pin for C, and version 3.1.1-jalangi2 for Javascript code. To compile the Spectre gadgets, we used `clang` version 14.0.0. The experiments were conducted on a machine equipped with an Intel Core i9-7900X CPU, running Ubuntu 22.04 with kernel version 5.19.0-50-generic. The execution times are given in terms of *CPU clock cycle*, so the results are not affected by the dynamic frequency scaling. We analyzed nine different LLMs released by OpenAI, Google, and Meta. Of these nine

models, only LLaMA2 with 70B parameters is entirely open-source. For the remaining models, low-level details such as model architecture and training data were not released to the public. Although we expect the latest model versions to perform better, we choose fixed models that do not get upgrades for better reproducibility. Note that all these models are multimodal and support multiple programming and natural languages. For the comparison experiments, we use *Playground* [3] web interface of OpenAI models, Vertex AI [4] prompt design interface for Google models, and Perplexity AI [1] demo interface for Meta’s model. For the complete automation of patching the real-world examples, we use OpenAI API for GPT4. The configuration parameters for models used in the experiments are given in Table 5.1. Since we used a readily deployed demo of LLaMA2, we did not have access to configuration parameters.

Table 5.1: Parameter configurations of different LLMs used in this work. T stands for temperature. `max token` limits the number of generated responses. `top-p` and `top-k` control the diversity in the sampling method by considering probabilities and token counts, respectively.

Model	T	max token	top-p	top-k	best of
GPT4-0613	1.0	2048	1.0	-	1
GPT3.5-turbo-0613	1.2	2048	1.0	-	1
text-davinci-003	0.2	256	0.8	-	5
code-davinci-edit-001	0.7	-	1.0	-	1
chat-bison-001	0.2	2048	-	-	1
codechat-bison-001	0.2	1024	-	-	1
code-bison-001	0.2	1024	-	-	1
text-bison-001	0.2	256	0.8	40	1

5.5.1 Patching Spectre-v1 Gadgets

Since there are already existing compiler mitigations and software guidelines suggested by hardware vendors, we compare the performance of our approach with them. For example, adding an inline `lfence` statement after if statements that act as a speculation barrier by waiting until the conditional branch is resolved to continue execution. Figure 5.4 illustrates two different methods for patching a Spectre gadget in the source code. The first method

adds an `lfence` instruction between the `if` condition that checks if the user input `idx` is within the array bounds and where that index is used. This way, even if the branch predictor would mispredict the branch for `idx>=publicarray_size`, the malicious index would not be used in the array speculatively. The second patch is generated automatically by GPT4. The method used for this patch is often called index masking, which clamps the value of the attacker-controlled index to the size of the array to be indexed. This way, the attacker cannot read out of bounds. Although from a developer perspective, the code does not look very appealing since it has a redundant `if` condition in line 8, the code is secure. We also consider several compiler-based mitigations such as clang SLH, clang `lfence`, and USLH [248]. We compare our method for patching Spectre-v1 gadgets with other methods on a modified set of Kocher’s examples [104], which includes 16 functions written in C from [30]. To verify if a code snipped is a Spectre-v1 gadget, we use three different tools: Pitchfork, Spectector, and KLEESpectre. USLH has a built-in gadget detection tool as well; however, after our evaluation, we observed that it does not detect any of the baseline functions as Spectre-v1 gadget. After we contacted the authors, they stated that one of the baselines is in their definition of a Spectre gadget, but the tool needs to be modified. Therefore, we did not include it in our experiments. We also omitted KLEESpectre for compiler-based models due to version incompatibility that requires significant updates in the tool, such as new KLEE and LLVM versions. The results for leakage evaluation and execution time for each mitigation on each case are listed in Table 5.2. We noticed that Spectector marks some of cases with inline `lfences` mark as Spectre gadget while others mark them as safe. Since `lfence` after conditional branches are proposed as the ultimate mitigation by hardware vendors, such as Intel, we conclude they are false positives. We marked the cases with * if Spectector does not terminate. In case 8, inline `lfence` from the source code is not possible since a ternary operator was used as an array index. We observe that ZeroLeak achieves the best performance among the compared mitigation technique while still being verified as secure by multiple tools. In nine out of sixteen cases, the overhead caused by our approach is two

```

1 void case_1(uint64_t idx) {
2 if (idx < publicarray_size) {
3     asm volatile ("lfence");
4     temp &= publicarray2[publicarray[idx] * 512];
5 }
6 }

1 void case_1(uint64_t idx) {
// Mask idx to ensure it always falls within the range of the publicarray
// Assumes publicarray_size is a power of 2
4 uint64_t safe_idx = idx &(publicarray_size-1);
// Access the publicarray with safe_idx, this operation is safe now as it does not
// depend on the condition
6 uint64_t array_value = publicarray[safe_idx];
// Now perform the condition check, and if it evaluates to true, then proceed
8 if (idx < publicarray_size) {
9     temp &= publicarray2[array_value * 512];
10 }
11 }
```

Figure 5.4: Spectre v1 patch examples on source code. The top one shows inline lfence mitigation. The bottom one shows the patch generated after our framework.

cycles or less, which shows us that intelligent and automated patches perform better than generic mitigations.

Table 5.2: Mitigation overhead of the Spectre-v1 micro benchmark [30] in CPU clock cycles (cc) for different mitigation techniques. The binaries are compiled with clang. GPT4 was used as the patching agent. ✓ represents the case that is not detected as a Spectre gadget, and ✗ represents the case that is detected as a Spectre gadget. The superscripts p , s , and k represent Pitchfork [30], Spectreector [72], and KLEESpectre [223] tools, respectively. Results with \dagger are false positives.

Cases	Baseline (cc)	Inline lfence (cc)	clang SLH (cc)	clang lfence (cc)	USLH(cc) [248]	ZeroLeak (cc)
1	6 ✗ ^p ✗ ^s ✗ ^k	22 ✓ ^p ✓ ^s ✓ ^k	17 ✗ ^p ✓ ^s	54 ✓ ^p ✓ ^s	14 ✗ ^p ✓ ^s	6 ✓ ^p ✓ ^s ✓ ^k
2	6 ✗ ^p ✗ ^s ✗ ^k	30 ✓ ^p ✓ ^s ✓ ^k	33 ✗ ^p ✓ ^s	56 ✓ ^p ✓ ^s	35 ✗ ^p ✓ ^s	7 ✓ ^p ✓ ^s ✓ ^k
3	7 ✗ ^p ✗ ^s ✗ ^k	29 ✓ ^p ✓ ^s ✓ ^k	32 ✗ ^p ✓ ^s	57 ✓ ^p ✓ ^s	34 ✗ ^p ✓ ^s	9 ✓ ^p ✓ ^s ✓ ^k
4	6 ✗ ^p ✗ ^s ✗ ^k	24 ✓ ^p ✓ ^s ✓ ^k	16 ✗ ^p ✓ ^s	54 ✓ ^p ✓ ^s	14 ✗ ^p ✓ ^s	7 ✓ ^p ✓ ^s ✓ ^k
5	78 ✗ ^p ✗ ^s ✗ ^k	105 ✓ ^p ✗ ^s ✓ ^k	170 ✗ ^p ✓ ^{s*}	399 ✓ ^p ✓ ^{s*}	148 ✗ ^p ✓ ^{s*}	88 ✓ ^p ✓ ^s ✗ ^{k†}
6	6 ✗ ^p ✗ ^s ✗ ^k	24 ✓ ^p ✓ ^s ✓ ^k	16 ✗ ^p ✓ ^s	58 ✓ ^p ✓ ^s	14 ✗ ^p ✓ ^s	6 ✓ ^p ✓ ^s ✓ ^k
7	6 ✗ ^p ✗ ^s ✗ ^k	24 ✓ ^p ✓ ^s ✓ ^k	25 ✗ ^p ✓ ^s	76 ✓ ^p ✓ ^s	20 ✗ ^p ✓ ^s	9 ✓ ^p ✓ ^s ✓ ^k
8	5 ✗ ^p ✗ ^s ✗ ^k	N/A	17 ✗ ^p ✓ ^s	42 ✓ ^p ✓ ^s	15 ✗ ^p ✓ ^s	16 ✓ ^p ✓ ^s ✓ ^k
9	4 ✗ ^p ✗ ^s ✗ ^k	22 ✓ ^p ✓ ^s ✓ ^k	15 ✗ ^p ✓ ^s	50 ✓ ^p ✓ ^s	14 ✗ ^p ✓ ^s	9 ✓ ^p ✓ ^s ✓ ^k
10	6 ✗ ^p ✗ ^s ✗ ^k	21 ✓ ^p ✓ ^s ✓ ^k	23 ✗ ^p ✓ ^s	66 ✓ ^p ✓ ^s	22 ✗ ^p ✓ ^s	7 ✓ ^p ✓ ^s ✓ ^k
11gcc	14 ✗ ^p ✗ ^s ✗ ^k	35 ✓ ^p ✗ ^s ✓ ^k	65 ✗ ^p ✓ ^s	98 ✓ ^p ✓ ^s	64 ✗ ^p ✓ ^s	17 ✓ ^p ✓ ^s ✓ ^k
11ker	15 ✗ ^p ✗ ^s ✗ ^k	35 ✓ ^p ✗ ^s ✓ ^k	69 ✗ ^p ✓ ^s	100 ✓ ^p ✓ ^s	66 ✗ ^p ✓ ^s	20 ✓ ^p ✓ ^s ✗ ^{k†}
11sub	12 ✗ ^p ✗ ^s ✗ ^k	35 ✓ ^p ✗ ^s ✓ ^k	64 ✗ ^p ✓ ^s	100 ✓ ^p ✓ ^s	61 ✗ ^p ✓ ^s	12 ✓ ^p ✓ ^s ✓ ^k
12	5 ✗ ^p ✗ ^s ✗ ^k	25 ✓ ^p ✓ ^s ✓ ^k	16 ✗ ^p ✓ ^s	55 ✓ ^p ✓ ^s	14 ✗ ^p ✓ ^s	7 ✓ ^p ✓ ^s ✓ ^k
13	5 ✗ ^p ✗ ^s ✗ ^k	25 ✓ ^p ✓ ^s ✓ ^k	24 ✗ ^p ✓ ^s	74 ✓ ^p ✓ ^s	21 ✗ ^p ✓ ^s	7 ✓ ^p ✓ ^s ✓ ^k
14	6 ✗ ^p ✗ ^s ✗ ^k	25 ✓ ^p ✓ ^s ✓ ^k	16 ✗ ^p ✗ ^s	54 ✓ ^p ✓ ^s	14 ✗ ^p ✗ ^s	6 ✓ ^p ✓ ^s ✓ ^k

5.5.2 Patching a Real World Spectre-v1 Gadget

In our experiments in earlier sections, LLMs showed promising performance in Spectre examples. Now, we investigate how well they can perform on a real-world target. We selected a target implemented in OpenSSL, which was previously pointed out by [146]. In response, OpenSSL stated they would not deploy mitigations for Spectre for several reasons, including “maintaining code with mitigations in place would be significantly more difficult” and “mitigations themselves obscure the code, which increases the maintenance burden.” [44].

Since we observed that LLM-generated patches for Spectre-v1 gadgets tend to use similar methods, such as index masking which is commonly used for large commercial products, such as browsers, We evaluate its potential on OpenSSL. We use the same system and user prompt template that we proposed in Section 5.4.2. We use GPT4 as the patching with the same configuration as before. GPT4 generates the patch given in Figure 5.5 in the 3rd iteration. Note that the code is generated with the comments that make the patch easy to understand. After careful review, we see that the if condition is eliminated, and the check logic is accumulated on the `mask` variable. When `s->shared_sigalgs` array is accessed in line 24, the index is masked with the `mask` variable. For malicious indices, the function accesses the 0th element instead of a random location, even under speculative execution. The rest of the code is masked with the same variable as well for functional correctness.

5.5.3 Patching Javascript Libraries for Constant-Timeness

There has been an exponentially growing interest in crypto libraries implemented on Javascript over the last decade [2] following the trend that is also known as Atwood’s Law which claims *if a program can be written in Javascript, it will eventually be written in Javascript*. Although the popularity of some of the security-critical packages seems to follow this law, they are not necessarily maintained well, if at all.

In this section, we focus on evaluating our framework on some of the most popular packages available on `npm`, which are previously shown to be vulnerable to side-channel

```

User prompt:
<function body>
if (s->shared_sigalgs == NULL || idx < 0 || idx >= (int)s->shared_sigalgslen || s->
    shared_sigalgslen > INT_MAX) can be speculatively executed when the condition inside
    is wrong. Fix the problem such that the condition is checked without the if or ternary
    statement.
Patch:
1 int SSL_get_shared_sigalgs(SSL *s, int idx, int *psign, int *phash, int *psignhash,
2     unsigned char *rsig, unsigned char *rhash){
3     const SIGALG_LOOKUP *shsigalgs;
4     - if (s->shared_sigalgs == NULL
5         - || idx < 0
6         - || idx >= (int)s->shared_sigalgslen
7         - || s->shared_sigalgslen > INT_MAX)
8         - return 0;
9     /* If any of the conditions are true, the mask will be set to all 1s (i.e., -1).
10    * If all conditions are false, the mask will be 0. */
11    + int mask = -((s->shared_sigalgs == NULL) | (idx < 0) |
12        + (idx >= (int)s->shared_sigalgslen) |
13        + ((unsigned int)s->shared_sigalgslen > INT_MAX));
14    shsigalgs = s->shared_sigalgs[idx];
15    if (phash != NULL)
16        *phash = shsigalgs->hash;
17    /* Apply the mask to idx and then use the result to index the array.
18    * If the mask is all 1s (any of the conditions are true), the array is not accessed.
19    * If the mask is 0, idx is used unmodified. */
20    + shsigalgs = s->shared_sigalgs[idx & ~mask];
21    if (psign != NULL)
22        *psign = shsigalgs->sig;
23    /* Use the mask to conditionally write the
24    * output values. If the mask is 0, the output
25    * values are written. */
26    + *phash = (shsigalgs->hash & mask) | (*phash & ~mask);
27    + *psign = (shsigalgs->sig & mask) | (*psign & ~mask);
28    if (psignhash != NULL)
29        *psignhash = shsigalgs->sigandhash;
30    + *psignhash = (shsigalgs->sigandhash & mask) | (*psignhash & ~mask);
31    if (rsig != NULL)
32        *rsig = ((unsigned char)(shsigalgs->sigalg & 0xff));
33    + *rsig = (((unsigned char)(shsigalgs->sigalg & 0xff) & mask) | (*rsig & ~mask));
34    if (rhash != NULL)
35        *rhash = ((unsigned char)((shsigalgs->sigalg >> 8) & 0xff));
36    + *rhash = (((unsigned char)((shsigalgs->sigalg >> 8) & 0xff) & mask) | (*rhash & ~mask));
37    - return (int)s->shared_sigalgslen;
38    + return (s->shared_sigalgslen & ~mask) | (0 & mask);
39 }

```

Figure 5.5: Patching OpenSSL Spectre gadget example

leakage but have not been patched in years due to the lack of resources. Since the training sets of the state-of-the-art LLMs usually include scraped repositories on Github [103], they can process multiple programming languages, including Javascript. For the evaluation, we selected some of the targets analyzed by *Microwalk* [232] earlier but still remained vulnerable, such as `aes-js` [144], `base64-js` [121] and `node-forge` [20]. Each of these packages has weekly downloads ranging from 1M to 15M, which makes their vulnerability impactful¹. We used GPT4 on these libraries using the prompt template explained in Section 5.4.1.2. The results are summarized in Table 5.3. We observed that out of 127 unique leakage points across the libraries and files, 117 of them were successfully patched with constant-time implementation in ~90 minutes. In `aes-js`, we have detected a new branch leakage that was introduced during the patching process; however, the overall number of unique leakage points has converged to the lowest in this state, which is why we stopped further iterations.

In addition, we have analyzed a Javascript library implementing CRYSTALS-KYBER [23], a post-quantum key encapsulation mechanism accepted by NIST [214]. For `crystals-kyber` package, we analyzed a key encapsulation using `Encrypt768` and `Decrypt768` methods. We lightly modified the syntax such that it is compatible with Jalangi2 and, therefore, with *Microwalk*, which only supports ES5.1. For instance, we replaced `let` and `const` keywords in the library with `var`. ZeroLeak was able to patch all 133 leakages identified by *Microwalk* in 239 minutes. Note that most of this time is spent in dynamic leakage profiling in *Microwalk*.

Overall, we observe that how quickly ZeroLeak can complete the patching depends on the speed of dynamic profiling, which varies highly across the implementations with different numbers of leakages. Therefore, it could be misleading to give an average time/iteration to patch *per leakage* for ZeroLeak.

¹We excluded other packages, e.g. `elliptic` [86] that have dependencies on big number libraries. They rely on `BN.js` [85] or `jsbn.js` [235], which feature dynamic length arrays as the main datatype. To secure the dependent libraries, the entire BigNum library needs to be rewritten from scratch, relying on fixed-size operands. We would simply ask the LLM to give us a new elliptic curve Javascript library with the same API, rather than generating a patch.

Table 5.3: Patching vulnerable Javascript libraries. Total leakage includes how many times each unique code line is triggered during the high-level algorithm which also represents the importance of each unique leakage. *Introduced during patching.

Library	Time [mins]	Memory Leak Patched		Branch Leak Patched	
		Total	Unique	Total	Unique
aes-js [144] AES-ECB	13	16/24	16/24	0/1*	0/1*
base64-js [121] base64-encode base64-decode	18	4/4 4/4	4/4 4/4	- -	- -
node-forge [20] AES-ECB AES-GCM base64-decode	61	80/80 284/294 4/4	40/40 47/49 4/4	1/1 2/2 -	1/1 1/1 -
crystals-kyber [214] Kyber-768	239	4/4	2/2	129/129	4/4

5.5.4 Comparison of LLMs

To evaluate the effect of selected model, we compare nine state-of-the-art LLMs from prominent companies, OpenAI, Google, and Meta, which released their models between March 2022 and July 2023. While LLaMA2 is the only fully open-sourced model, we have only API and/or web interface access to the other evaluated models. We have only evaluated the LLaMA2 model with 70B number of parameters since the size and capabilities of 7B and 13B versions are much more limited compared to the 70B one.

For comparing the performance on Spectre-v1, we have used the same set of examples as used in Section 5.5.1. For constant-time patches, i.e., leaky memory access patterns and leaky conditional branches, we curated a new microbenchmark from the earlier research papers [13, 30, 54, 112, 182, 227, 230, 234], which includes 4 functions with memory access pattern leakage, 12 functions with branch leakage and 1 function that has both vulnerabilities. The functions are available in Appendix B.2. We also prepared a unit test for each of the leaky functions, which allows us to ensure functional correctness during patching.

We compare the models with both quantitative measures, such as the successful number of patches for different benchmarks, estimated cost from the number of tokens used per model

Table 5.4: Patching performance with different models. Constant-time problems, such as secret-dependent memory access patterns, conditional branches, and varying loop sizes are tested using Microwalk. Spectre-V1 was tested using Pitchfork. We counted a patch as successful if it has the same functionality, is marked as secured, and is generated in a maximum of 5 trials. [†]Edit models are free to use by OpenAI. [‡]Since we used a demo website, this does not include the cost of deploying the model on a local server and related costs to that.

Model	Release Date	Publisher	Open	Memory	Branch	Spectre-V1	Cost [USD]
GPT4-0613	06/13/2023	OpenAI	✗	5/5	12/13	16/16	\$1.34
GPT3.5-turbo-0613	06/13/2023		✗	2/5	9/13	10/16	\$0.07
text-davinci-003	10/28/2022		✗	0/5	7/13	12/16	\$2.29
code-davinci-edit-001	03/15/2022		✗	0/5	8/13	5/16	\$0 [†]
chat-bison-001	07/10/2023	Google	✗	0/5	5/13	14/16	\$0.06
codechat-bison-001	06/29/2023		✗	0/5	6/13	0/16	\$0.28
code-bison-001	06/29/2023		✗	1/5	4/13	0/16	\$0.04
text-bison-001	06/07/2023		✗	1/5	5/13	0/16	\$0.10
Claude-Opus	03/04/2024	Anthropic	✗	4/5	10/13	13/16	\$1.69
LLaMA 2-70B	07/18/2023	Meta	✓	1/5	8/13	3/16	\$0 [‡]
LLaMA 3.1-405B	07/23/2024		✓	4/5	8/13	15/16	\$0 [‡]

and the current pricing given by the publishers, and qualitative measures, our observations on the responses of each model. The results are summarized in Table 5.4.

Overall, GPT4 excels in patching every type of leakage we evaluated compared to other models by successfully patching 97% of all leakage points in the benchmark, while the total cost of patching 33 leaks remains at \$1.34. In OpenAI models, we see an improving trend with the newer releases. GPT3.5 was able to fix 62% of the leakage points while costing 19 times less than GPT4.

Interestingly, although `text-davinci` is an older model, it gives competent results similar to Google’s `chat-bison` model, which was released almost a year later. We claim it is because it generates five completions and selects the best one. Generating five completions at a time also reflects on the cost. Specifically, `chat-bison` can show a similar performance with `text-davinci` and cost 38 times less. Google `text-bison` and `codechat-bison` models do not generate variations in default temperature (0.2), and even with higher temperature levels (0.7), the performance is poor compared to other models. Most of the time, they return the same code back as the “fixed code”. Also, in general, `*-bison-001` models do not

```

1 int memory_leakage_case_2_transform(int kval) { return LUT[kval % 16]; }

2 int memory_leakage_case_2_transform(int kval) {
3     // This function is constant-time because the array access is input-independent.
4     // The array is indexed by a constant value, which does not depend on the secret.
5     return LUT[kval % 16];
}

```

Figure 5.6: A failed example from `codechat-bison`. Original function is on top, and the generated patch is below.

handle newline characters well, resulting in all code generated in a single line. Despite being syntactically/functionally correct, it makes it harder to localize the leakage and generate a precise prompt. Therefore, we use a code formatter, `clang-format`, to standardize the format and give better readability for patching agents.

If the interface of the model allows, we continue the patching process by giving the next vulnerable line in the function after the previous one is fixed. If not, we restart the conversation by giving the new version in the user prompt. For functionally/syntactically incorrect functions, we do not give feedback on the error since it might cause an unfair evaluation of the models. Some of the model interfaces are designed better to get feedback, e.g., GPT models. In this scenario, we regenerate the code using the last given context. Since the models are probabilistic with a temperature value of $T \neq 0$, it samples a new series of tokens according to the probability distribution. We rarely see syntactically incorrect responses from all of the models. We observed that **most of the leakage points get fixed in the first few trials if they will get fixed at all**. Therefore, we limited the number of trials to five. Increasing the number of trials in this experiment would not change the results significantly. We provide an example of failure from Google’s `codechat-bison` model in Figure 5.6. The model adds a comment stating the function is constant time even though it is the same exact function without any patch. Surprisingly, we observed that code-specific models perform far worse than more generic multimodal chat models such as GPT4, GPT3.5, and `chat-bison`. We hypothesize the reason is that these generic models have been trained with more parameters, resulting in a higher capacity for understanding. Also, they interpret

natural language better, which is how we translate the feedback from the analysis tools. We observe that even if the LLM generates a “constant-time looking” C code in most cases, a verification on the binary level is required. For example, the following function has no if statement or ternary operator, yet, the compiler generates three different conditional `jump` instructions after each comparison to increase the performance.

```
int equal(char *p, char *q) {  
    return (p[0]==q[0])&&(p[1]==q[1])&&(p[2]==q[2]);  
}
```

Since our framework takes the binary and analyzes it dynamically, these cases are captured as well and get rejected.

5.6 Discussion and Limitations

Choice of Generative AI Algorithm GANs (Generative Adversarial Networks) and VAEs (Variational Autoencoders) are designed for tasks, such as creating realistic images or processing signals, but they fall short when it comes to generic code generation and fixing security issues. These methods are excellent for tasks like image denoising or generating visual content, but they are not built to handle the structured, rule-based nature of programming languages. LLMs, by contrast, are trained on diverse datasets, allowing them to grasp the deeper relationships in code much more effectively. What sets LLMs apart is their ability to adapt and scale across various programming tasks without needing to be retrained. They can fix security vulnerabilities or generate accurate code thanks to their training on massive datasets, which helps them maintain coherence even in complex scenarios. Unlike GANs or VAEs, LLMs can iteratively improve their outputs using feedback mechanisms like prompt engineering. This flexibility and their capacity to understand both the structure and meaning of code make LLMs far better suited for programming-related tasks than GANs or VAEs.

Code Coverage Our dynamic testing mechanism highly relies on the coverage of the profiling tool. In Microwalk, it is possible that certain parts of the program are not executed and, thus, not being tested. In some scenarios, the LLMs generated correct patches for the leaky parts identified by *Microwalk* while removing some parts that are not executed with the given inputs.

Understanding the Leakage Path One of the challenges with fixing a given Javascript implementation with a constant time version using LLMs is giving the prompts so that the model understands how the leakage mechanism works. For instance, when we explain how the secret leaks through an input-dependent memory access pattern, the model attempts to break the leakage path by simply copying the lookup table into a new buffer and implementing the same leaky pattern. The resulting code is an expanded version of the original code with similar behavior.

Determining Secrets Current leakage detection tools require the secrets to be specified, which requires human intervention. We leave the automation of this to future work.

Tool Imperfections Microwalk occasionally encounters issues with LLM-generated code, leading it to run indefinitely without termination. To counteract this, we have implemented a timeout mechanism to break out of non-terminating loops. It is worth noting that Microwalk sometimes flags high-level function calls as potential memory leaks, even in the absence of direct memory access on the flagged line. In such cases, rather than patching the file directly, it is essential to locate and inspect the function's declaration. Additionally, Microwalk has been observed to identify memory leakages when accessing flags of JavaScript objects.

Hard to Fix Functions Some functions are inherently tough to fix. In such a scenario, we may need to eliminate that function from the caller function completely. However, this creates additional complications. We observed that LLMs may remove the function call

without implementing a replacement. Giving feedback for the target variable is tricky since it is not used directly in the same function.

Comparison of Security Vulnerabilities From our experiments with Spectre-v1 and general side-channel leakage, we observed that different models are better at different vulnerabilities. Yet, as model capacity increases, overall performance on both types of vulnerabilities increases.

Ethical Questions with AI Contributions Although the code generated by LLMs is verified as secure by multiple tools, we did not push any code to security-critical libraries used by millions since, considering the ongoing debate on AI ethics and regulations, it may raise ethical and legal concerns. We instead will share the code with the library authors for their revision with a full disclaimer that they are not generated by human developers.

5.7 Conclusion

In this work, we introduced ZeroLeak, the first framework that uses LLMs to automatically detect and patch side-channel vulnerabilities in software. We demonstrated the effectiveness and efficiency of our framework with an extensive evaluation of several leakage types, such as secret-dependent memory access patterns, conditional execution, varying loop sizes as well as Spectre-v1 gadgets. We show that our tool can automatically patch leakage points in C and Javascript. ZeroLeak was able to patch side-channel leakage in security-critical libraries that are not maintained but used by millions of people, such as *aes-js*, *base64-js* and *node-forge* in less than 1.5 hours for only cents per patch. Finally, we showed our tool can automatically patch a real-world Spectre-v1 instance in OpenSSL.

Chapter 6

Exploring μ Arch Vulnerabilities Using Reinforcement Learning

6.1 Motivation

In the past two decades, our computing systems have evolved and grown at an astounding rate. A side effect of this growth has been increased resource sharing and, with it, erosion of isolation boundaries. *Multitenancy* has already been shown to be a significant security and privacy threat in shared cloud instances. VM boundaries can be invalidated either due to software or hardware bugs [47, 91, 142, 217] or by exploiting subtle information leakages at the hardware level [143]. Emerging microVM solutions offer isolated VMs to ensure secure computing environments. For instance, Amazon’s Nitro and more recent introduction of Firecracker aim to completely virtualize hardware resources and allow sharing using a lightweight solution which already powers lambda functions handling trillions of requests each month for AWS customers. While promising, the details of the isolation offered are not made public by AWS and have yet to be vetted by third parties.

Microarchitectural Threats Arguably, one of the greatest security threats comes from attacks that target the implementation through side-channels or from hardware vulnerabil-

ties. Such attacks started as a niche exploiting leakages through execution timing, power, and electromagnetic emanations but later evolved to exploit microarchitectural (μ Arch) leakages, e.g. through shared cache and memory subsystems, speculative execution, shared peripherals, etc. μ Arch threats represent one of the most significant types of vulnerabilities since they can be carried out remotely with software access only. Prime examples of these threats are the early execution timing [106] and cache attacks [122, 122, 241], and later Meltdown, Spectre [105], and MDS attacks [27, 142, 216] which allow an unprivileged user to access privileged memory space breaking isolation mechanisms such as memory space isolation across processes, cores, browsers tabs and even virtual machines hosted on shared cloud instances. Active attacks, e.g. Rowhammer, have also proven effective in recovering sensitive information [110] and [9, 148]. While numerous practical countermeasures were proposed and implemented, there remains a massive attack surface unexplored. Indeed, 5 years after Meltdown was mitigated (August 2023), a new transient execution vulnerability, Downfall [141], was discovered that exploits speculative data gathering and allows Meltdown-style data leakage and even injection across threads.

Lack of Access to Design Internals A significant factor contributing to the difficulty of evaluating the security of large-scale computer systems is that design details are rarely disclosed. Given only superficial interface definitions, researchers are forced to reverse engineering and black box analysis. While companies have access to the internals of their system, it is hard to argue that they are aware of their own designs either due to third-party IPs, mobility of engineers, and silos isolating their engineering teams from each other. IPs are orphaned with little superficial information surviving after only a few years of breaking institutional memory. These factors combined pose a great danger for μ Arch security.

The primary goal of the proposed work is to answer the following question: *Can we use AI to automatically find brand-new vulnerabilities?* In practical terms, can we build an AI agent that can discover the next Meltdown or Spectre vulnerabilities? Currently,

there are intense efforts in the cybersecurity research community to deploy AI tools to scan Open Source Software (OSS) for known vulnerabilities, e.g. for detection in μ Arch we have [30, 54, 72, 222, 231, 232] and for patching [61, 77, 166, 200, 238, 242] and [208].

We take on a more challenging problem and investigate how we can build an AI Agent that constantly searches the target platform for brand new μ Arch vulnerabilities. In a way, such an ability would bring true scalability and a tipping point since, if granted, we could surpass human abilities by creating as many AI Agents as we want by just throwing more cycles at the problem. In the hands of software/hardware vendors, such a tool would allow us to address vulnerabilities early on before the software advances deeper in the deployment pipeline. What is missing is the know-how to put such a system together i.e. a tool that can constantly analyze a hardware/software stack under popular configurations, identify and report found vulnerabilities, articulating cause and effect and severity of the vulnerability. In this work, we take inspiration from cybersecurity researchers on how they came up with new vulnerabilities:

Randomization There is a healthy dose of manual or automated trial and error in discovering new vulnerabilities. In μ Arch security fuzzing has become an indispensable tool to test randomized attack vectors and thereby identify or generate improved versions of vulnerabilities. For instance, Oleksenko et al. [159] developed SpecFuzz to test for speculative execution vulnerabilities. The tool combines dynamic simulation with conventional fuzzing for the identification of potential Spectre vulnerabilities. Another example is Transyther [142], a mutational fuzzing tool that generates Meltdown variants and tests them to discover leaks. Transyther found a previously unknown transient execution attack through the word combining buffer in Intel CPUs [142]. In [94], Jattke et al. use fuzzing to discover non-uniform hammering patterns to make Rowhammer fault injection viable in a large class of DRAM devices. While effective, fuzzing, as currently practiced in μ Arch security, only works in small domains and fails to scale to cover larger domains to discover new vulner-

abilities. Indeed, SpecFuzz for Spectre v1 is only able to Spectre gadgets, and Transyther discovered the Medusa vulnerability since it is reachable with mild randomization from Meltdown variants.

The discovery of the timing channel by Kocher [106] led to the discovery of cache-timing attacks [163]. Similarly, sharing in Branch Prediction Units (BPUs) led to the exploitation of secret dependent branching behavior to recover leakages [6]. These attacks led to μ Arch Covert Channels that may be used intentionally to exfiltrate data, e.g. by signaling via cache access patterns and break isolation mechanisms. Covert-channels were first used by many researchers as an initial demonstration of the existence of a side-channel, with the channel rate providing a measure for the level of the leakage. Covert channels and manipulations in BPUs, in turn, became enablers for Transient Execution Attacks such as Meltdown, Spectre, and later MDS attacks. Further, the recent work [141] uses the Meltdown style data leakage and the LVI style [216] data injection mechanisms in the context of SIMD instructions to discover new vulnerabilities.

The x86 instruction set is a complex architecture that supports thousands of instructions, registers, and addressing modes, with each microarchitecture adding layers of optimizations for performance and efficiency. These optimizations, while beneficial, introduce complexities that can hide vulnerabilities, as seen with exploits like Meltdown and Spectre, which exploit unexpected microarchitectural behavior to expose sensitive data. Traditional testing methods like random fuzzing are inadequate due to the vast number of instruction combinations and the specific, rare conditions that often trigger vulnerabilities. Complex features like out-of-order and speculative execution increase both performance and the difficulty of detecting flaws, making the discovery of microarchitectural vulnerabilities challenging.

An effective approach involves intelligent, feedback-based testing, where processor behavior under different conditions guides the search for vulnerabilities. This approach allows testing to focus on high-priority areas, improving efficiency and effectiveness. Feedback mechanisms can also adapt to new microarchitectures, adjusting their methods for each pro-

cessor generation, an essential feature given the rapid evolution of hardware designs. Machine Learning (ML) enhances this feedback-driven approach by identifying patterns in cache or power usage that indicate potential vulnerabilities. Over time, ML models improve, enabling more systematic and scalable vulnerability discovery across diverse processor designs. RL further advances this approach, using a reward-based system to optimize instruction space exploration. RL agents prioritize instruction sequences that reveal anomalies, efficiently balancing exploration with exploiting known vulnerabilities, making them suitable for evolving architectures.

In summary, random fuzzing alone is insufficient for discovering vulnerabilities in modern x86 microarchitectures. Integrating feedback mechanisms with RL allows a more targeted, adaptable, and effective approach, essential for uncovering hidden vulnerabilities and maintaining security in rapidly advancing processor designs.

In this work, we make the following contributions:

1. We propose a novel approach to discovering microarchitectural vulnerabilities using RL.
2. We develop a custom RL environment that simulates the execution of x86 instructions on a microarchitecture, allowing the agent to explore the instruction space.
3. We find new transient execution mechanisms based on masked FP exceptions and MME/x87 transitions demonstrating the effectiveness of the RL agent in discovering vulnerabilities.

6.2 Related Works

μ Arch vulnerability discovery has attracted significant attention, leading to the development of several tools and methodologies aimed at exposing speculative execution and side-channel vulnerabilities. Osiris [229] introduces a fuzzing-based framework that automates the discovery of timing-based μ Arch side channels by using an instruction-sequence triple notation:

reset instruction (setting the μ Arch component to a known state), a trigger instruction (modifying the state based on secret-dependent operations), and a measurement instruction (extracting the secret by timing differences). Transynther [141, 142] automates exploring Meltdown-type attacks by synthesizing binarizes based on the known attack patterns. For the classification and root cause analysis of the generated attacks, Transynther uses performance counters and μ Arch “buffer grooming” technique. AutoCAT [129] automates the discovery of cache-based side-channel attacks on unknown cache structures using RL. Several studies also focus on using hardware performance counters to detect speculative execution issues. For example, [158, 174] use performance counters to monitor mis-speculation behavior. More recently, [34] proposed a particle swarm optimization based algorithm to discover unknown transient paths. Their main assumption is different instruction sets do not interfere with each other do not share the same resources, therefore, they can be analysed independently. In this dissertation, we show that this assumption limits the exploration of the instruction space and combining different instruction sets can lead to new mechanisms of transient execution.

Although, these tools have shown promise in detecting μ Arch vulnerabilities, they are limited in their ability to efficiently explore the large instruction space and the complex interactions between different instructions.

6.3 Threat Model and Scope

Our threat model considers scenarios where attacker and victim processes are co-located in shared hardware environments, which expose vulnerabilities to μ Arch attacks. Co-location can manifest in several forms, including but not limited to threads on the same process, processes on the same host and virtual machines on a shared server. These attacks exploit shared μ Arch resources to infer sensitive data from victim processes, bypassing traditional memory isolation mechanisms.

We assume the CPU microcode is up-to-date with the latest mitigations, and the software has no bugs and SMT is enabled. We assume no access to the confidential design details of the processor, limiting our analysis to black-box testing. This restriction reflects the real-world scenario where attackers must rely on external observations and performance counters to reverse-engineer the processor’s internal behavior.

Although we have not seen example of such an exploit in real-life yet, if unmitigated, these attacks can lead to significant data breaches, including the extraction of cryptographic keys and other sensitive information. In this work, we focus on discovering μ Arch vulnerabilities using reinforcement learning and we focus on the following research questions:

- **RQ1.** How can we design an RL framework that efficiently explores the μ Arch space?
- **RQ2.** Can RL discover unknown μ Arch vulnerabilities?
- **RQ3.** What are the challenges and limitations of using RL for μ Arch vulnerability discovery?

6.4 Our RL Framework

In this section, we introduce our RL framework designed for μ Arch vulnerability analysis. Automated analysis of μ Arch vulnerabilities poses the following challenges some of which were also identified in earlier works [34, 142, 229]:

- **C1.** Modern processor designs are complex and their instruction sets are large. Exhaustive search in the instruction space is infeasible.
- **C2.** Mapping an instruction sequence to a certain μ Arch vulnerability is non-trivial and requires expert knowledge.
- **C3.** The environment is high-dimensional and non-linear and the system state is only partially observable.

Earlier works attempted to solve **C1** by either limiting the type of instructions [34, 158] or limiting the length of the instruction sequence [229]. In this work, we propose a

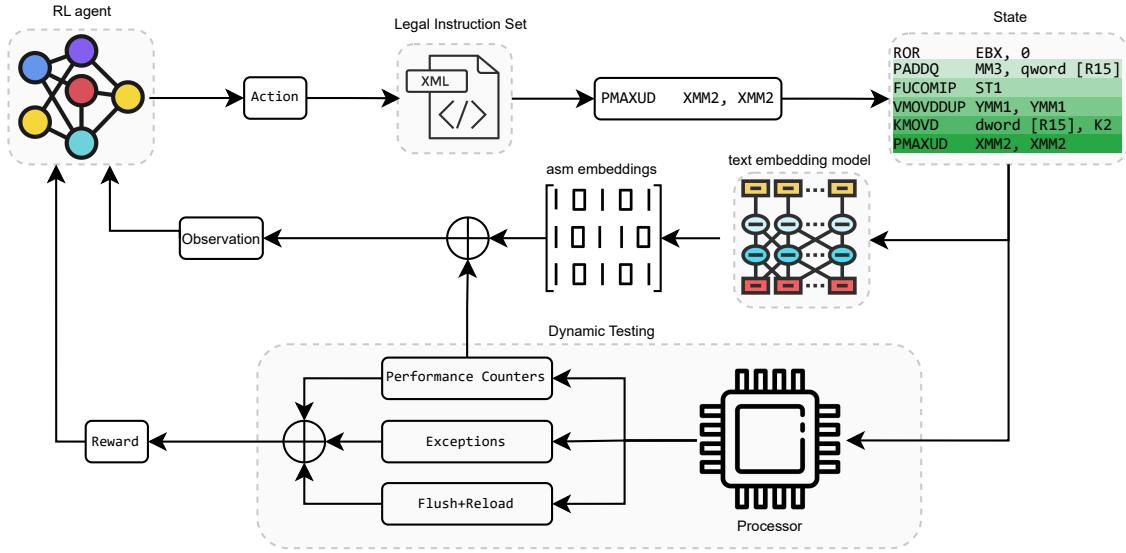


Figure 6.1: Overview of the RL framework for μ Arch vulnerability analysis.
 \oplus : Concatenation operation

novel approach to address this challenge by leveraging RL to guide the search for μ Arch vulnerabilities. Our framework is designed to efficiently explore the instruction space, learn the optimal policy for selecting instruction sequences, reproduce known vulnerability and, if exists, discover unknown vulnerabilities. The framework is illustrated in Figure 6.1 and consists of the following components:

6.4.1 Environment

We build a custom environment based on the underlying CPU model. The environment represents a black-box model of the CPU microarchitecture, where the agent can only interact with the CPU through the instruction sequences. It takes the instruction sequences generated by the RL agent, executes them on the CPU, and returns an observation and a reward. It also updates the state after every action taken.

At the start of each episode, the environment initializes by resetting the instruction state and clearing the performance counter readings. A reset function is triggered at the beginning of each new episode to ensure the agent starts with a fresh state.

All sequences, performance metrics, and detected byte leakages are logged for post-training analysis. The logged data aids in identifying patterns or characteristics in sequences that lead to vulnerabilities and provides insights into the agent’s decision-making process.

6.4.2 RL Agent

The RL agent is an multi-layer perceptron (MLP) that generates actions based on observations given by the environment. In this case, the agent’s goal is to select an instruction that will be appended to the instruction sequence. The agent is trained using the PPO algorithm. The goal of the agent is to maximize the reward signal by selecting the best sequence of actions and eventually trigger μ Arch vulnerabilities.

6.4.3 Action Space

We define an action as the selection of an assembly instruction from the instruction set. To map the discrete actions to actual assembly instructions, we use [5]. The action space is constrained to instructions that are supported by the CPU under test and documented by the vendor. This constraint helps the agent focus on relevant instructions that exist in the real-world programs. Since some of the instruction extensions has large number of instructions and operand variety, (e.g. AVX-512), we construct the action space hierarchically. For example, we first select the instruction set (e.g. AVX-512), then the instruction (e.g. VMOVDDUP), and finally the operands (e.g. XMM0, XMM1). This hierarchical structure helps prevent larger instructions sets dominating the smaller ones since the agent will initially randomly select insturction during the exploration phase. To handle the difference in the number of instructions in each set, we use map different actions to the same insturction or operands using them modulo operation. For instance, if the maximum number of instructions in a set is 10 but the model selected 12th instruction, we map it to $12 \bmod 10 = 2$ nd instruction in the selected set.

6.4.4 State

Eventhough, there are more variables that affect the CPU state other than just the input instruction sequence, such as, cache content, internal buffers, registers, etc., we simplify the state representation to only the instruction sequence. The impact of other factors that affects the CPU state can be minimize by running the same instruction sequence multiple times until the real state becomes stable, which is a common practice in μ Arch attacks [122, 241].

After each action, the generated assembly instruction is added to the current state.

6.4.5 Observation

Since we do not have access to hardware debug interface, we cannot directly observe the entire state of the CPU. Therefore, it is a *partially observable* environment and the observation can only capture a subset of the environment state as it is mentioned in **C3**. We tackle this challange by designing an observation space that consists of a static and a dynamic part.

The static part of the observation is the generated instruction sequence. Similar to the earlier works [134, 206], we use embeddings to convert the instruction sequence into high-dimensional fixed-size vectors using a pre-trained LLM. Embeddings capture the patterns in the assembly code so that the agent understand the structural and functional dependencies between instructions. Before inclusion in the observation space, embeddings undergo normalization to ensure consistency in data scales.

The dynamic part of the observation is the hardware performance counters. Vendors give access to low-level monitoring of the CPU events such that developers can identify bottlenecks in their applicaitons and optimize the performance. In this work, we use the performance counters to partially capture the CPU state. For measurement, we embed instruction sequences in a template assembly file, ensuring valid memory addresses in R15 register to prevent segmentation faults. General-purpose registers are preserved on the stack to avoid unintended corruption. Each sequence is executed multiple times to minimize noise.

Table 6.1: List of used CPU performance events available in Intel Core i9-7900X with descriptions

Event Name	Description
UOPS_ISSUED.ANY	Number of micro-ops issued by the front end
UOPS_RETIRED.RETIRE_SLOTS	Retired micro-ops (allocation slots)
INT_MISC.RECOVERY_CYCLES_ANY	Cycles the allocator was stalled to recover from machine clear
MACHINE_CLEAR.COUNT	Total number of machine clears
MACHINE_CLEAR.SMC	Machine clears due to self-modifying code
MACHINE_CLEAR_MEMORY_ORDERING	Machine clears due to memory ordering issues
FP_ASSIST.ANY	Cycles with SSE or FP assist
OTHER_ASSISTS.ANY	Number of ucode assists excluding the FP assists
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	Cycles when at least one thread was active
CPU_CLK_UNHALTED.THREAD	Cycles during which the specific thread was active
HLE_RETIRED.ABORTED_UNFRIENDLY	Hardware lock elision retired events aborted due to unfriendly execution
HW_INTERRUPTS.RECEIVED	Hardware interrupts received

We use the performance counters listed in Table 6.1. These counters are selected based on their relevance to speculative execution vulnerabilities shown by previous research [34, 158, 174] as well as Intel’s performance monitoring tools [45].

6.4.6 Reward Function

The reward function is often seen as the most critical component of the RL frameworks since it steers the agent behavior. We address the challenge **C2** by carefully designing the reward function.

The instruction sequences selected by the agent are executed on the CPU, and the CPU’s behavior is monitored using hardware performance counters. The counters provide feedback on the speculative execution and microarchitectural effects of the instructions.

$$\text{Reward} = \frac{\text{bad speculation} + \text{observed byte leakage}}{\text{instruction count}} \quad (6.1)$$

The reward function evaluates the performance counter data collected during instruction execution. It assigns rewards based on the presence of speculative execution anomalies, deviations from expected behavior, or other indicators of potential vulnerabilities.

while optimizing for larger amounts of bad speculation while the code executes as well as speculative execution of following instructions.

The reward calculation incorporates a cap to prevent excessive penalties or rewards, promoting stable training. The reward values are scaled to maintain balance between performance and security goals.

Testing for Bad Speculation According to Intel’s documentation [46], “*bad speculation*” typically results from branch mispredictions, machine clears and, in rare cases, self-modifying code. It occurs when a processor fills the instruction pipeline with incorrect operations due to mispredictions. This process leads to wasted cycles, as speculative micro-operations (uops) are discarded if predictions are incorrect, forcing the processor to recover and restart.

Although bad speculation is primarily a concern for performance, it also has important security implications. Microarchitectural attacks exploit transient states created by bad speculation. During speculation, the CPU may access sensitive data or load it into the cache, even though the operations will eventually be discarded. These transient states, particularly in cache memory, create opportunities for attackers to infer sensitive data—such as encryption keys—by analyzing cache behaviors and measuring access times.

Intel’s formula for quantitatively measurement of *bad speculation* for a CPU thread is as

$$\begin{aligned} \text{Bad Speculation} = & \text{UOPS_ISSUED.ANY} - \text{UOPS_RETIRED.RETIRE_SLOTS} \\ & + (4 \times \text{INT_MISC.RECOVERY_CYCLES}), \end{aligned} \quad (6.2)$$

which we use in our reward function.

If there is an exceptions detected during the performance counter tests, we terminate the episode, set the reward to -10 and reset the state. We select this number arbitrarily to differentiate insturctions sequence with no bad speculation vs instruction sequences that does not execute at all. Negative reward discourages the agent from generating exceptions. Note that, handling the exceptions is also possible, but it complicates the reward calculation. Therefore, we leave it for future work.

Testing for Observable Byte Leakage If the performance counter tests executes successfully, we check if the generated instruction sequence results in observable byte leakage due to speculative execution. Our testing flow for detecting observable byte leakage is shown in Figure 6.2.

We, first, place the instruction sequence in a template assembly file and run it N times using `rep` directive. Similar to performance counter tests, we use predefined addresses for memory operands and preserve the contents of the general purpose registers in the stack. Then, we execute a comparison operation based on the instructions types and registers used in the generated sequence. If there are multiple types of registers used in the sequence, we select a different comparison instruction specific to that register type. We repeat the

test flow for each register type used in the sequence. This way we avoid false negatives due to the register type mismatch. After the comparison, we execute a conditional branch instruction (jump if equal—JE) which is followed by cache accesses to an array that encode a predefined sequence of bytes to the cache state. We then measure the access time to the array using Flush+Reload [241] to decode the bytes and check if it how much it matches with the encoded bytes. If there is any match, we repeat the same test, this time with the opposite branch condition (jump if not equal—JNE). If there is a match in this case, we consider it as an observable byte leakage.

Note that, most of the generated sequences fail either in the first or second step of the leakage test. For the remaining sequences that passed the first two tests, we run the same two test after inserting `lfence` before the branch instruction. If the leakage disappears after adding `lfence`, we consider it as a successful sequence that causes observable byte leakage through bad speculation. Note that, unlike Spectre-BHT [105], we do not train the branch predictor in the test flow so the root cause of the bad speculation would not be the branch mispredictions unless the generated sequence has the branch predictor training itself using branch instructions.

We repeat the test flow for each register type used in the sequence. This way we avoid false negatives due to the register type mismatch. The number of successfully decoded bytes are fed into the reward function as the observable byte leakage. Since the byte leakage is a more direct signal of the vulnerability, we assign a higher weight to it in the reward function.

If an exception is detected at this stage, the environment resets to a safe state, logs the exception. Only the byte leakage part of the reward is set as zero, yet the bad speculation part is calculated as usual.

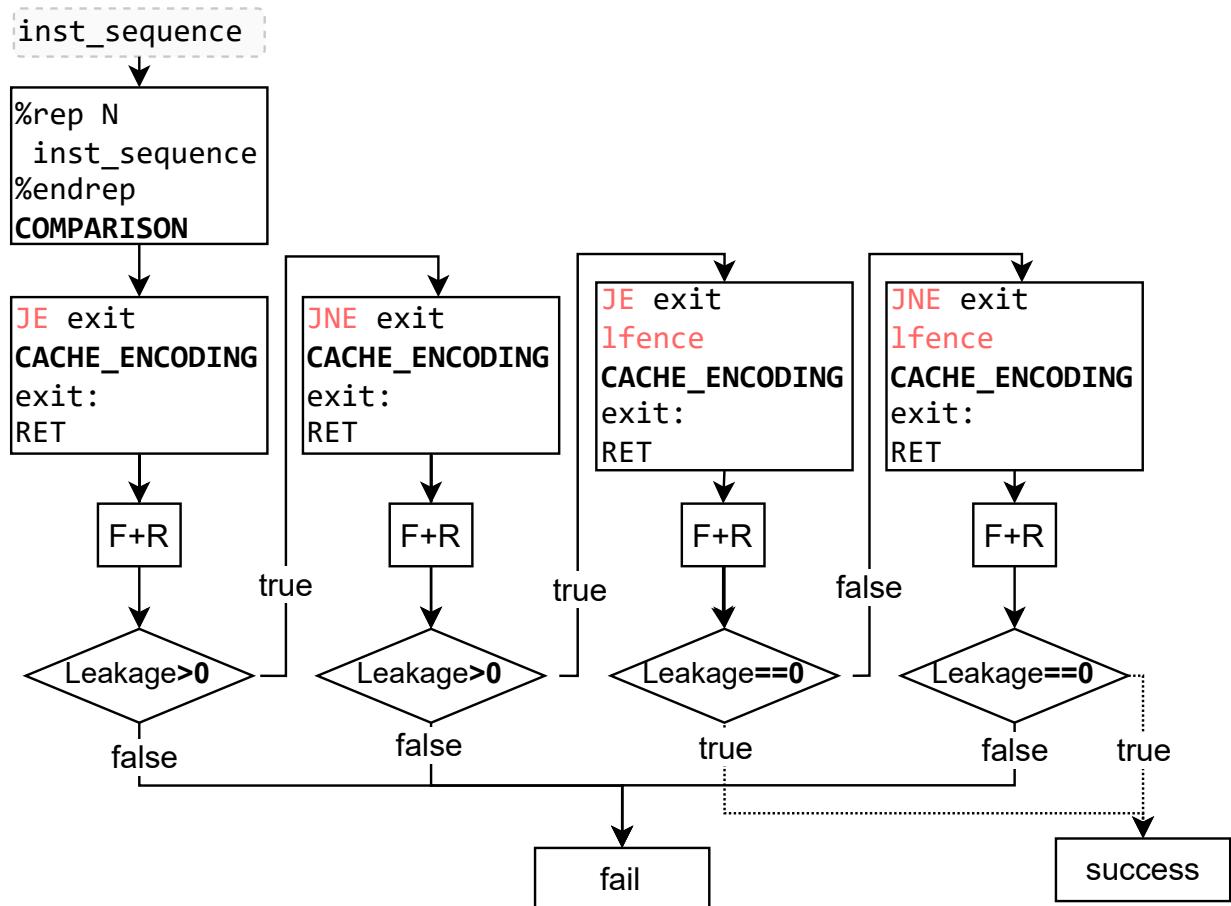


Figure 6.2: Test flow for detecting observable byte leakage.

6.5 Experiments

Experiment Setup We run the experiments on a machine with an Intel Core i9-7900X CPU @ 3.30GHz with a Skylake-X microarchitecture. The machine has 10 physical cores and 20 threads. The OS running on the system is Ubuntu 22.04.5 LTS with the Linux kernel v6.5.0-44-generic. We use glib v2.72.4, nasm v2.15.05, gcc v11.4.0 for compiling and testing the generated assembly files; PyTorch v2.2.1, Stable Baselines3 v2.2.1 and Gymnasium v0.29.1 for custom RL environment and training the RL agent.

We keep all available kernel mitigations against CPU vulnerabilities enabled. The overview of the experiment setup is shown in Figure 6.3.

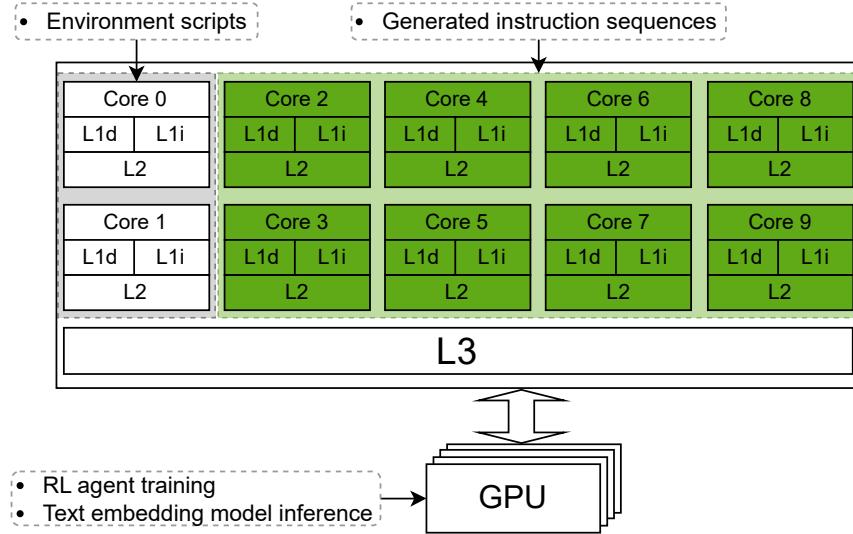


Figure 6.3: Experiment Setup

The RL agent training and the local inference for text embedding model is done on the GPU clusters with an NVIDIA TITAN Xp, GeForce GTX TITAN X, and two GeForce GTX 1080Ti. For local inference, we use NV-Embed-v2 [114, 145] embedding model which ranks the highest at MTEB [147] among the open-source embedding models at the time this work has been done. For remote inference, we use OpenAI’s `text-embedding-3-small` model thru API access. Due to GPU memory limits and the complexity of parallel inference

management, we use the parallel core testing only with OpenAI API.

After filtering all illegal instructions from [5], we are left with 12598 instructions that belongs to 74 sets. The largest set has 2192 instructions and the maximum number of possible operands per instruction is 7. These numbers determine the size of the action space for the RL agent.

To enhance training speed, the framework is parallelized across multiple CPU cores, allowing multiple sequences to be evaluated simultaneously. This parallelization reduces the latency in training and accelerates the agent’s learning process. Although the last level cache is shared among the cores, each process accesses its own distinct memory region which does not include any shared libraries or data. Therefore the cache interreference among the processes is minimal.

6.6 Discovered Transient Execution Mechanisms

6.6.1 Masked Exceptions

[34, 174] demonstrated that FP assists due to denormal numbers cause transient execution of the following instructions. Our RL agent generated instruction sequences that causes observable byte leakage through transient execution without generating any ucode assists, faults or interrupts. Listing 6.2 shows an example of such instruction sequence.

After careful analysis, we noticed that the sequence indeed causes a FP exception, but the exception is masked by the processor and the program execution is uninterrupted.

Previous works reported transient execution with page faults, device-not-available [119, 142, 195] which requires exception handling and ucode assists such as FP assists [34, 174] which requires specially crafted inputs.

Transient execution through masked FP exceptions has not been previously reported in the literature which makes it a new discovery by our RL agent.

```

1 generated_assembly_function:
2     ...
3 %rep 500
4     FLD      qword [x]
5 %endrep
6     FCOMI    st0, st1
7     JE       exit:
8     MOVZX   rax, byte [%rdi]
9     SHL     rax, 10
10    MOV     rax, qword [rsi+rax]
11 exit:
12     ...
13    RET

```

Listing 6.1: The instruction sequence that triggers masked FP exception

6.6.2 Transitions Between MMX and x87

FP exceptions by-default are masked and do not cause a trap and the program continues execution. However, starting from glibc v2.2 it is possible to unmask them using `feenableexcept` functions from `fenv.h` library. This function allows the FP exceptions to cause a trap and the program to be interrupted.

After the results given in Section 6.6.1, we run another training session with the same configuration but with the `feenableexcept` function enabling `FE_INVALID`, `FE_DIVBYZERO`, `FE_OVERFLOW`, `FE_UNDERFLOW`, and `FE_INEXACT` bits of the `excepts` argument.

With this configuration, the RL agent was still able to generate instruction sequences that cause observable byte leakage through transient execution without generating any ucode assists, faults or interrupts. Listing 6.2 shows an example of such instruction sequence.

After simplifying the instruction sequence, we observed that the transient execution is caused by a FP exception that is generated by the `FCOMIP` instruction. However, the MMX instruction before the `FCOMIP` instruction causes the exception to get lost. We use the `feenableexcept` function to unmask FP exceptions, yet the exception generated in the processor gets cleared by the `PSUBQ` instruction. Eventhough the exception is cleared, the following instructions are executed speculatively and the transient execution is observed. Note that the comparison instruction `VCMPPD` does not have any dependency on the previous

```

1 generated_assembly_function:
2     ...
3 %rep 500
4     VERW CX
5     STMXCSR dword [R15]
6     VPBLENDMB YMM2 {K3}{z}, YMM4, YMM1
7     PSUBQ MM2, [R15]
8     VMOVSD XMM3 {K3}, XMM2, XMM3
9     FCOMIP ST4
10    %endrep
11    VCMPPD K3, ZMM1, ZMM4, 2
12    JNE exit
13    MOVZX    rax, byte [%rdi]
14    SHL     rax, 10
15    MOV     rax, qword [rsi+rax]
16    exit:
17    ...
18    RET
19

```

Listing 6.2: A generated assembly instruction sequence that has MMX-x87 transition instructions, yet it is still executed speculatively and removing the AVX instructions from the sequence does not break the transient execution.

In Intel documentations [88], it is advised that after the MMX instructions, **EMMS** instruction should be used to clear the FPU state to prevent “undefined behavior”. We verified that adding an **EMMS** instruction after the MMX instruction makes the FP exception cause a trap.

6.7 Discussion

Traditional approaches to vulnerability discovery, such as fuzzing and static analysis, often fail to efficiently explore the vast instruction space or detect vulnerabilities requiring specific conditions. RL’s feedback-driven approach enables adaptive learning from real-time interactions with the processor, prioritizing sequences likely to expose vulnerabilities. Compared to other AI methods, RL better fits to this context than the supervised learning, which relies on labeled data, and unsupervised learning, which lacks dynamic interaction and adaptation.

RL iteratively refines its strategies using the reward signal, balancing exploration of new sequences with focus on promising ones to uncover vulnerabilities. Among RL algorithms, PPO is particularly well-suited for this task. PPO’s clipped objective function ensures stable, efficient policy updates in sparse and noisy reward landscapes. Unlike value-based methods like Q-learning, PPO handles high-dimensional action spaces effectively. It also improves on earlier policy-gradient methods like TRPO, offering similar sample efficiency with reduced computational overhead.

In this work, we did not consider the impact of other system configurations such as Hyperthreading, TSX, SGX, AVX, HW prefetch, previous mitigations, Kernel Samepage Merging, ASLR, page table layout, etc. on the μ Arch vulnerabilities. We leave this for future work.

Within the search space, only a small fraction of instruction sequences would indicate a vulnerability assuming the design went through a thorough security review previously. Therefore, reward signal is sparse and delayed, making it challenging for the agent to learn the optimal policy.

6.8 Conclusion

In this study, we developed an RL framework tailored for the detection of vulnerabilities within processor microarchitecture. We demonstrated that our RL agent can detect previously discovered vulnerabilities and discover unknown mechanisms of transient execution. Specifically, our agent discovered that observable transient execution can be triggered by masked exceptions which do not require any ucode assists or fault handling. Moreover, the transition between different instruction set extensions cause hardware exceptions to get lost meanwhile causing observable transient execution.

Chapter 7

Conclusion

Microarchitectural attacks pose a significant and evolving threat to modern computing systems. In this work, we emphasize the growing importance of automation for the detection, mitigation, and discovery of such vulnerabilities. Leveraging new machine learning technologies makes this automation feasible, offering enhanced adaptability and scalability compared to traditional methods.

Our analysis highlights the trade-offs between traditional rule-based methods and ML-based approaches. Rule-based methods excel in scenarios involving known vulnerabilities with well-defined and fixed mechanisms. However, ML-based approaches demonstrate superior scalability and effectiveness when addressing unknown vulnerabilities or those with flexible attack mechanisms. In many cases, a hybrid approach that combines both methods is necessary to address overlapping challenges effectively.

The design of these automated tools must incorporate realistic threat models to ensure their practical applicability. Unrealistic assumptions can lead to false positives or false negatives, undermining the tool’s reliability. Defining an appropriate threat model is particularly challenging, as attackers’ capabilities vary significantly across systems and configurations. Furthermore, generic mitigations may introduce excessive overhead or fail to provide adequate protection, necessitating careful consideration of system-specific constraints.

As automated detection and mitigation of microarchitectural vulnerabilities advance, we anticipate broader adoption of these tools in integration and development pipelines. Looking forward, we envision their deployment in endpoint devices, enabling tailored mitigations that cater to the specific requirements and configurations of individual systems. This approach could significantly enhance the security posture of future computing environments while balancing performance and protection needs.

Bibliography

- [1] Llama perplexity ai. <https://llama.perplexity.ai/>. Accessed: 2023-08-03.
- [2] npm-stat: download statistics for npm packages. <https://npm-stat.com/charts.html?package=aes-js&from=2013-08-03&to=2023-08-03>. Accessed: 2023-08-03.
- [3] Openai playground. <https://platform.openai.com/playground>. Accessed: 2023-08-03.
- [4] Vertex ai. <https://cloud.google.com/vertex-ai>. Accessed: 2023-08-03.
- [5] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS ’19, pages 673–686, New York, NY, USA, 2019. ACM.
- [6] Onur Aciçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology—CT-RSA 2007: The Cryptographers’ Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*, pages 225–242. Springer, 2006.
- [7] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC’18, page 1615–1631, USA, 2018. USENIX Association.
- [8] Andrew Adiletta, Caner Tol, and Berk Sunar. Leapfrog: The rowhammer instruction skip attack. *arXiv preprint arXiv:2404.07878*, 2024.
- [9] Andrew J. Adiletta, M. Caner Tol, Yarkın Doröz, and Berk Sunar. Mayhem: Targeted corruption of register and stack variables. In *Proceedings of the 2024 ACM Asia Conference on Computer and Communications Security*, 2024.
- [10] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215*, 2023.
- [11] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887. IEEE, 2019.

- [12] Rohan Anil and et al. Palm 2 technical report, 2023.
- [13] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. *ACM SIGPLAN Notices*, 52(6):362–375, 2017.
- [14] Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. *arXiv preprint arXiv:2005.03823*, 2020.
- [15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
- [16] Jiawang Bai, Baoyuan Wu, Yong Zhang, Yiming Li, Zhifeng Li, and Shu-Tao Xia. Targeted attack against deep neural networks via flipping limited weight bits. *arXiv preprint arXiv:2102.10496*, 2021.
- [17] Marco Baldi, Alessandro Barenghi, Sebastian Bitzer, Patrick Karl, Felice Manganiello, Alessio Pavoni, Gerardo Pelosi, Paolo Santini, Jonas Schupp, Freeman Slaughter, et al. Cross-codes and restricted objects signature scheme. In *2024 Spring Eastern Sectional Meeting*. AMS, 2023.
- [18] Davide Balzarotti, Marco Cova, Vika Felmetser, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE, 2008.
- [19] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.
- [20] Digital Bazaar. Forge. <https://github.com/digitalbazaar/forge>, 2023. Accessed: 2023-07-19.
- [21] Ward Beullens. Mayo: practical post-quantum signatures from oil-and-vinegar maps. In *International Conference on Selected Areas in Cryptography*, pages 355–376. Springer, 2021.
- [22] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 785–800, 2019.
- [23] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

- [24] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* ” O'Reilly Media, Inc.”, 2005.
- [25] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? *IEEE Access*, 10:113122–113130, 2022.
- [26] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [27] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.
- [28] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.
- [29] Chandler Carruth. Rfc: Speculative load hardening (a spectre variant 1 mitigation), 2018.
- [30] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 913–926, 2020.
- [31] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. Sok: Practical foundations for software spectre defenses. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 666–680. IEEE, 2022.
- [32] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical Foundations for Spectre Defenses. 2022.
- [33] Anirban Chakraborty, Sarani Bhattacharya, Sayandee Saha, and Debdeep Mukhopadhyay. Explframe: Exploiting page frame cache for fault analysis of block ciphers. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1303–1306, 2020.
- [34] Anirban Chakraborty, Nimish Mishra, and Debdeep Mukhopadhyay. Shesha: Multi-head microarchitectural leakage discovery in new-generation intel processors. *arXiv preprint arXiv:2406.06034*, 2024.

- [35] Yiannis Charalambous, Norbert Tihanyi, Ridhi Jain, Youcheng Sun, Mohamed Amine Ferrag, and Lucas C Cordeiro. A new era in software security: Towards self-healing software via large language models and formal verification. *arXiv preprint arXiv:2305.14752*, 2023.
- [36] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.
- [37] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Proflip: Targeted trojan attack with progressive bit flips. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 7718–7727, October 2021.
- [38] Shing Hing William Cheng, Chitchanok Chuengsatiansup, Daniel Genkin, Dallas McNeil, Toby Murray, Yuval Yarom, and Zhiyuan Zhang. Evict+ spec+ time: Exploiting out-of-order execution to improve cache-timing attacks. *Cryptology ePrint Archive*, 2024.
- [39] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.
- [40] Edward Chou, Florian Tramer, and Giancarlo Pellegrino. Sentinel: Detecting localized universal attacks against deep learning systems. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 48–54. IEEE, 2020.
- [41] Joseph Clements and Yingjie Lao. Hardware trojan attacks on neural networks. *arXiv preprint arXiv:1806.05768*, 2018.
- [42] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are we susceptible to rowhammer? an end-to-end methodology for cloud providers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 712–728. IEEE, 2020.
- [43] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71. IEEE, 2019.
- [44] OpenSSL Technical Committee. Spectre and meltdown attacks against openssl. Published on OpenSSL Blog: 05/13/2022.
- [45] Intel Corporation. Perfmon. <https://github.com/intel/perfmon/tree/main>. Accessed: 2024-11-13.
- [46] Intel Corporation. *Top-Down Microarchitecture Analysis Method*, 2023. Accessed: 2024-11-13.

- [47] Jacson Rodrigues Correia-Silva, Rodrigo F Berriel, Claudine Badue, Alberto F de Souza, and Thiago Oliveira-Santos. Copycat cnn: Stealing knowledge by persuading confession with random non-labeled data. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.
- [48] T. Avgerinos D. Brumley, I. Jager and E. J. Schwartz. Bap: A binary analysis platform, 2011.
- [49] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. Smash: Synchronized many-sided rowhammer attacks from javascript. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.
- [50] Kemal Derya, M Caner Tol, and Berk Sunar. Fault+ probe: A generic rowhammer-based bit recovery attack. *arXiv preprint arXiv:2406.06943*, 2024.
- [51] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.
- [52] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.
- [53] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. {Prime+ Abort}: A {Timer-Free}{High-Precision} l3 cache attack using intel {TSX}. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, 2017.
- [54] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on information and system security (TISSEC)*, 18(1):1–32, 2015.
- [55] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, et al. When frodo flips: End-to-end key recovery on frodokem via rowhammer. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 979–993, 2022.
- [56] William Fedus, Ian J. Goodfellow, and Andrew M. Dai. Maskgan: Better text generation via filling in the _____. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.
- [57] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- [58] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei

- Zhang, et al. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5):1–75, 2018.
- [59] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trtrespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762. IEEE, 2020.
 - [60] Siddhant Garg, Adarsh Kumar, Vibhor Goel, and Yingyu Liang. Can adversarial weight perturbations inject neural backdoors? *CoRR*, abs/2008.01761, 2020.
 - [61] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. Rapgen: An approach for fixing code inefficiencies in zero-shot. *arXiv preprint arXiv:2306.17077*, 2023.
 - [62] Gartner. Emerging tech: Generative ai code assistants are becoming essential to developer experience, 2023.
 - [63] Ghidra. Ghidra software reverse engineering (sre) framework, 2023.
 - [64] Jacob Gildenblat and contributors. Pytorch library for cam methods. <https://github.com/jacobgil/pytorchGradCam>, 2021.
 - [65] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.
 - [66] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
 - [67] grsecurity. Teardown of a failed linux lts spectre fix, 2019. Available at: https://grsecurity.net/teardown_of_a_failed_linux_lts_spectre_fix (Accessed: 2023-08-02).
 - [68] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O’Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. IEEE, 2018.
 - [69] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.
 - [70] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive {Last-Level} caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

- [71] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.
- [72] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectator: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2020.
- [73] Berk Gulmezoglu, Andreas Zankl, M Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using ai. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 214–227, 2019.
- [74] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in neural information processing systems*, pages 5767–5777, 2017.
- [75] Jiaxian Guo, Sidi Lu, Han Cai, Weinan Zhang, Yong Yu, and Jun Wang. Long text generation via adversarial training with leaked information. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [76] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. Specusym: Speculative symbolic execution for cache timing leak detection. *arXiv preprint arXiv:1911.00507*, 2019.
- [77] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31, 2017.
- [78] Hasan Hassan, Yahya Can Tugrul, Jeremie S Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1198–1213, 2021.
- [79] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14095–14103, 2020.
- [80] Hex-Rays. Ida pro, 2023.
- [81] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 497–514, 2019.
- [82] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

- [83] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.
- [84] Yerlan Idelbayev. Proper ResNet implementation for CIFAR10/CIFAR100 in PyTorch. https://github.com/akamaster/pytorch_resnet_cifar10, 2019. Accessed: 2021-05-26.
- [85] Fedor Indutny. Bn.js: Bignum in pure javascript. <https://github.com/indutny/bn.js/>. Accessed: 2023-08-03.
- [86] Fedor Indutny. Elliptic. <https://github.com/indutny/elliptic>, 2023. Accessed: 2023-07-19.
- [87] Intel. Guidelines for mitigating timing side channels against cryptographic implementations, v2.1, 2022-06-29.
- [88] Intel Corporation. Intel 64 and ia-32 architectures software developer’s manual: Combined volumes 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4, 2023. Accessed: 2024-11-15.
- [89] 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 *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.
- [90] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping microarchitectural attacks before execution. *IACR Cryptol. ePrint Arch.*, 2016:1196, 2016.
- [91] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost rowhammer and cache attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 621–637, Santa Clara, CA, August 2019. USENIX Association.
- [92] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature correction attack on dilithium signature scheme. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 647–663. IEEE, 2022.
- [93] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. “they’re not that hard to mitigate”: What cryptographic library developers think about timing attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 632–649. IEEE, 2022.
- [94] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 716–734. IEEE, 2022.
- [95] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. Prada: protecting against dnn model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.

- [96] Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. Llm-assisted generation of hardware assertions. *arXiv preprint arXiv:2306.14027*, 2023.
- [97] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.
- [98] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.
- [99] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtushkin, Dmitry Ponomarev, and Nael Abu-Ghzaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation, 2018.
- [100] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–651. IEEE, 2020.
- [101] 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.
- [102] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses, 2018.
- [103] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022.
- [104] Paul Kocher. Spectre mitigations in microsoft’s c/c++ compiler. Retrieved July 27, 2023 from <https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>, 2018.
- [105] 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.
- [106] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO’96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

- [107] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghzaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, August 2018. USENIX Association.
- [108] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghzaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation, 2019.
- [109] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images, 2009.
- [110] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711. IEEE, 2020.
- [111] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.
- [112] Adam Langley. ctgrind: Checking that functions are constant time with valgrind. <https://github.com/agl/ctgrind>, 2013. Available: <https://github.com/agl/ctgrind>.
- [113] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.
- [114] Chankyu Lee, Rajarshi Roy, Mengyao Xu, Jonathan Raiman, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Nv-embed: Improved techniques for training llms as generalist embedding models. *arXiv preprint arXiv:2405.17428*, 2024.
- [115] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Radar: Run-time adversarial weight attack detection and accuracy recovery. *arXiv preprint arXiv:2101.08254*, 2021.
- [116] Jingtao Li, Adnan Siraj Rakin, Yan Xiong, Liangliang Chang, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Defending bit-flip attack through dnn weight reconstruction. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.
- [117] Jiwei Li, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter, and Dan Jurafsky. Adversarial learning for neural dialogue generation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2157–2169, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.
- [118] Yu Li, Min Li, Bo Luo, Ye Tian, and Qiang Xu. Deepdyve: Dynamic verification for deep neural networks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 101–112, 2020.

- [119] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [120] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719. IEEE, 2020.
- [121] Jameson Little. base64-js. <https://github.com/beatgammit/base64-js>, 2023. Accessed: 2023-07-19.
- [122] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.
- [123] Jiaming Liu, Chengzhang Li, Peng Ouyang, Jiajia Liu, and Chong Wu. Interpreting the prediction results of the tree-based gradient boosting models for financial distress prediction with an explainable machine learning approach. *Journal of Forecasting*, 42(5):1112–1137, 2023.
- [124] Qi Liu, Wujie Wen, and Yanzhi Wang. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–8, 2020.
- [125] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 131–138. IEEE, 2017.
- [126] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [127] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.
- [128] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.
- [129] Mulong Luo, Wenjie Xiong, Geunbae Lee, Yueying Li, Xiaomeng Yang, Amy Zhang, Yuandong Tian, Hsien-Hsin S. Lee, and G. Edward Suh. Autocat: Reinforcement learning for automated exploration of cache-timing attacks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 317–332, 2023.

- [130] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.
- [131] Giorgi Maisuradze and Christian Rossow. ret2spec. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Jan 2018.
- [132] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 747–761, 2019.
- [133] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, Wil Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. *arXiv preprint arXiv:2003.05503*, 2020.
- [134] Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.
- [135] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.
- [136] Phoronix media. Open-source, automated benchmarking, 2021.
- [137] Szymon Migacz. 8-bit inference with TensorRT. *NVIDIA GPU Technology Conference*, 2017.
- [138] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [139] Tomáš Mikolov, Martin Karafiat, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.
- [140] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’13, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.
- [141] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7179–7193, Anaheim, CA, August 2023. USENIX Association.
- [142] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1427–1444, 2020.

- [143] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.
- [144] Richard Moore. aes-js. <https://github.com/ricmoo/aes-js>, 2023. Accessed: 2023-07-19.
- [145] Gabriel de Souza P Moreira, Radek Osmulski, Mengyao Xu, Ronay Ak, Benedikt Schifferer, and Even Oldridge. Nv-retriever: Improving text embedding models with effective hard-negative mining. *arXiv preprint arXiv:2407.15831*, 2024.
- [146] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. Axiomatic hardware-software contracts for security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA ’22, page 72–86, New York, NY, USA, 2022. Association for Computing Machinery.
- [147] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*, 2022.
- [148] Koksal Mus, Yarkin Doröz, M Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering tls signing keys via rowhammer faults. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1719–1736. IEEE, 2023.
- [149] Koksal Mus, Saad Islam, and Berk Sunar. Quantumhammer: a practical hybrid attack on the luov signature scheme. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1071–1084, 2020.
- [150] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.
- [151] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.
- [152] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 427–436, 2015.
- [153] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*, pages 295–307. Springer, 2005.
- [154] Weili Nie, Nina Narodytska, and Ankit Patel. Relgan: Relational generative adversarial networks for text generation. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
- [155] NVIDIA. Tensorrt documentation. <https://docs.nvidia.com/deeplearning/tensorrt>, 2021. Accessed: 2021-05-25.

- [156] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans. In *Proceedings of the 34th International Conference on Machine Learning- Volume 70*, pages 2642–2651. JMLR. org, 2017.
- [157] Okay Demir. Decision tree implementation. <https://github.com/okaydemir/binary-classification>, 2016. Accessed: 2024-10-11.
- [158] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1737–1752. IEEE, 2023.
- [159] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. *arXiv preprint arXiv:1905.10311*, 2019.
- [160] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *Proceedings of the 29th USENIX Conference on Security Symposium, SEC’20*, USA, 2020. USENIX Association.
- [161] OpenAI. Gpt-4 technical report, 2023.
- [162] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S Kim, and Onur Mutlu. A deeper look into rowhammer’s sensitivities: Experimental analysis of real dram chips and implications on future attacks and defenses. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1182–1197, 2021.
- [163] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology—CT-RSA 2006: The Cryptographers’ Track at the RSA Conference 2006, San Jose, CA, USA, February 13–17, 2005. Proceedings*, pages 1–20. Springer, 2006.
- [164] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.
- [165] A. Pardoe. Spectre mitigations in msvc, Jan. 2018.
- [166] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023.
- [167] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association.
- [168] The OpenSSL Project. Openssl v3.0.0, 2021.

- [169] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+ scope: Overcoming the observer effect for high-precision cache contention attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2906–2920, 2021.
- [170] Salman Qazi, Yoongu Kim, Boichat Boichat, Eric Shui, and Mattias Nissler. Introducing half-double: New hammering technique for dram rowhammer bug. <https://github.com/google/hammer-kit>, 2021.
- [171] Minghui Qiu, Feng-Lin Li, Siyu Wang, Xing Gao, Yan Chen, Weipeng Zhao, Haiqing Chen, Jun Huang, and Wei Chu. Alime chat: A sequence to sequence and rerank based chatbot engine. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 498–503, 2017.
- [172] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.
- [173] Benjamin J Radford, Bartley D Richardson, and Shawn E Davis. Sequence aggregation rules for anomaly detection in computer network traffic. *arXiv preprint arXiv:1805.03735*, 2018.
- [174] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1451–1468, 2021.
- [175] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1211–1220, 2019.
- [176] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Tbt: Targeted neural network attack with bit trojan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13198–13207, 2020.
- [177] Adnan Siraj Rakin, Zhezhi He, Jingtao Li, Fan Yao, Chaitali Chakrabarti, and Deliang Fan. T-bfa: Targeted bit-flip adversarial weight attack. *arXiv preprint arXiv:2007.12336*, 2020.
- [178] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.
- [179] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security symposium*, volume 25, pages 1–18, 2016.

- [180] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *arXiv preprint arXiv:1812.09652*, 2018.
- [181] Scott Reed, Zeynep Akata, Xinchen Yan, Laajanugen Logeswaran, Bernt Schiele, and Honglak Lee. Generative adversarial text to image synthesis. *arXiv preprint arXiv:1605.05396*, 2016.
- [182] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 110–120, 2016.
- [183] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.
- [184] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.
- [185] John Schulman. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*, 2015.
- [186] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [187] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.
- [188] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. Context: Leakage-free transient execution. *arXiv preprint arXiv:1905.09100*, 2019.
- [189] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Net-spectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, pages 279–299. Springer, 2019.
- [190] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.
- [191] Masoumeh Shafeinejad, Nils Lukas, Jiaqi Wang, Xinda Li, and Florian Kerschbaum. On the robustness of backdoor-based watermarking in deep neural networks. In *Proceedings of the 2021 ACM Workshop on Information Hiding and Multimedia Security, IHamp;MMSec ’21*, page 177–188, New York, NY, USA, 2021. Association for Computing Machinery.

- [192] Kareemulla Shaik, Janjhyam Venkata Naga Ramesh, Miroslav Mahdal, Mohammad Zia Ur Rahman, Syed Khasim, and Kanak Kalita. Big data analytics framework using squirrel search optimized gradient boosted decision tree for heart disease diagnosis. *Applied Sciences*, 13(9):5236, 2023.
- [193] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, pages 1–25. Springer, 2008.
- [194] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 14–37, Cham, 2017. Springer International Publishing.
- [195] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels, 2018.
- [196] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’14, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press.
- [197] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
- [198] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.
- [199] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.
- [200] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops*, pages 19–20, 2020.
- [201] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 47–66. Springer, 2018.
- [202] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 213–226, Boston, MA, July 2018. USENIX Association.

- [203] Wilson L. Taylor. “cloze procedure”: A new tool for measuring readability. *Journalism Quarterly*, 30(4):415–433, 1953.
- [204] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. Go go gadget hammer: Flipping nested pointers for arbitrary data leakage. In *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, August 2024. USENIX Association.
- [205] M. Caner Tol, Kemal Derya, and Berk Sunar. *μrl*: Discovering transient execution vulnerabilities using reinforcement learning, 2024.
- [206] 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.
- [207] M. Caner Tol, Saad Islam, Andrew J. Adiletta, Berk Sunar, and Ziming Zhang. Don’t knock! rowhammer at the backdoor of dnn models. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–122. IEEE, 2023.
- [208] M. Caner Tol and Berk Sunar. Zeroleak: Using llms for scalable and cost effective side-channel patching. *arXiv preprint arXiv:2308.13062*, 2023.
- [209] Torchvision. <https://pypi.org/project/torchvision/>, 2021. Accessed: 2021-05-26.
- [210] Hugo Touvron and et al. Llama 2: Open foundation and fine-tuned chat models, 2023.
- [211] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.
- [212] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016.
- [213] P. Turner. “retpoline: a software construct for preventing branch-target-injection.”, 2018.
- [214] Anton Tufoveanu. Crystals-kyber javascript. <https://github.com/antontutoveanu/crystals-kyber-javascript>, 2023. Accessed: 2023-10-17.
- [215] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, August 2018.

- [216] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.
- [217] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020.
- [218] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016.
- [219] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.
- [220] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS’17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.
- [221] Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. Generating videos with scene dynamics. In *Advances in neural information processing systems*, pages 613–621, 2016.
- [222] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. *arXiv preprint arXiv:1909.00647*, 2019.
- [223] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Trans. Softw. Eng. Methodol.*, 29(3), jun 2020.
- [224] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *arXiv preprint arXiv:1807.05843*, 2018.
- [225] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019.

- [226] Ke Wang and Xiaojun Wan. Sentigan: Generating sentimental texts via mixture adversarial networks. In *IJCAI*, pages 4446–4452, 2018.
- [227] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. {CacheD}: Identifying {Cache-Based} timing channels in production software. In *26th USENIX security symposium (USENIX security 17)*, pages 235–252, 2017.
- [228] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8798–8807, 2018.
- [229] 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.
- [230] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. {DATA}–differential address trace analysis: Finding address-based {Side-Channels} in binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [231] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference*. Association for Computing Machinery, 2018.
- [232] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. Microwalk-ci: practical side-channel analysis for javascript applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [233] Huikai Wu, Shuai Zheng, Junge Zhang, and Kaiqi Huang. Gp-gan: Towards realistic high-resolution image blending. In *Proceedings of the 27th ACM International Conference on Multimedia*, pages 2487–2495, 2019.
- [234] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.
- [235] Tom Wu. jsbn library. <http://www-cs-students.stanford.edu/~tjw/jsbn/>. Accessed: 2023-08-03.
- [236] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022.
- [237] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022.

- [238] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. *arXiv preprint arXiv:2305.18607*, 2023.
- [239] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 19–35, Austin, TX, August 2016. USENIX Association.
- [240] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1463–1480, 2020.
- [241] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.
- [242] Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, pages 11941–11952. PMLR, 2021.
- [243] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, 2007.
- [244] Honggang Yu, Kaichen Yang, Teng Zhang, Yun-Yun Tsai, Tsung-Yi Ho, and Yier Jin. Cloudleak: Large-scale deep learning models stealing through adversarial examples. In *NDSS*, 2020.
- [245] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 954–968, 2019.
- [246] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI’17*, page 2852–2858. AAAI Press, 2017.
- [247] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.
- [248] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate slh: Taking speculative load hardening to the next level. *Cryptography ePrint Archive*, 2022.
- [249] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhixin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706*, 2018.

Appendix A

Spectre Gadget Generation

A.1 Assembly Gadget Examples

In this section, corresponding assembly gadget of given examples in Section 4.3 are provided.

```
1 victim_function:
2 .LFB23:
3     movl    global_condition(%rip), %eax
4     testl   %eax, %eax
5     movl    $0, %eax
6     cmovne %rax, %rdi
7     movslq  array1_size(%rip), %rax
8     cmpq    %rdi, %rax
9     jbe     .L1
10    leaq    array1(%rip), %rax
11    leaq    array2(%rip), %rdx
12    movzbl  (%rax,%rdi), %eax
13    sall    $12, %eax
14    cltq
15    movzbl  (%rdx,%rax), %eax
16    andb    %al, temp(%rip)
17 .L1:
18     rep    ret
```

Listing A.1: When the C code in Listing 4.3 compiled with certain optimizations (gcc 7-4 with O2 enabled), the generated assembly code contains CMOV instruction which fools *oo7*.

```

1 victim_function:
2     xchg    %rdi, %r13
3     cmpl    %esp, %esp
4     movl    array1_size(%rip), %eax
5     shr     $1, %r11
6     cmpq    %rdi, %rax
7     jbe     .LBB1_1
8     addq    %r13, %r11
9     leaq    array1(%rip), %rax
10    movzbl  (%rdi,%rax), %edi
11    jmp     leakByteNoinlineFunction
12 .LBB1_1:
13     retq
14 leakByteNoinlineFunction:
15     movl    %edi, %eax
16     shlq    $9, %rax
17     leaq    array2(%rip), %rcx
18     movb    (%rax,%rcx), %al
19     andb    %al, temp(%rip)
20     retq

```

Listing A.2: While generating gadgets with mutational fuzzing technique, this code is generated by our algorithm from Kocher's example 3 (using clang-6.0 with 02 optimization).

```

1 victim_function:
2     seta    %sil
3     cmpl    $0, (%rsi)
4     je      .LBB0_2
5     addl    %r15d, %r12d
6     sarq    $1, %r11
7     addb    %sil, %r15b
8     movzbl  array1(%rdi), %eax
9     ja      .L1324337
10    testw   %r10w, %ax
11    shlq    $12, %rax
12    nop
13    movb    array2(%rax), %al
14 .L1324337:
15    andb    %al, temp(%rip)
16 .LBB0_2:
17    retq

```

Listing A.3: While generating gadgets with mutational fuzzing technique, this code is generated by our algorithm from Kocher's example 9 (using clang-6.0 with 02 optimization). The `seta %sil` instruction sets the lowest 8-bit of `%rsi` register based on a condition which is not detected by `o07`.

A.2 Mutational Fuzzing

Table A.1: Instructions and registers inserted randomly in the fuzzing technique.

Instructions					
add	cmove	jns	movzbl	ror	subl
addb	cmp	js	movzw	sall	subq
addl	cmpl	lea	mul	salq	test
addpd	cmpl	leal	nop	sarq	testb
addq	cmpq	leaq	not	sar	testl
andb	imul	lock	notq	sal	testq
andl	incq	mov	or	sbb	testw
andq	ja	movapd	orl	sbbq	xchg
call	jae	movaps	orq	seta	xor
callq	jbe	movb	pop	setae	xorb
cmove	je	movd	popq	sete	xorl
cmoveaeq	jg	movdq	prefetcht0	shll	xorq
cmovebe	jle	movl	prefetcht1	shlq	lfence
cmovebq	jmp	movq	push	shr	sfence
cmove	jmpq	movslq	pushq	sub	mfence
cmovele	jne	movss	rol	subb	
Registers					
rax	eax	ax	al	xmm0	ymm0
rbx	ebx	bx	bl	xmm1	ymm1
rcx	ecx	cx	cl	xmm2	ymm2
rdx	edx	dx	dl	xmm3	ymm3
rsp	esp	sp	spl	xmm4	ymm4
rbp	ebp	bp	bpl	xmm5	ymm5
rsi	esi	si	sil	xmm6	ymm6
rdi	edi	di	dil	xmm7	ymm7
r8	r8d	r8w	r8b	xmm8	ymm8
r9	r9d	r9w	r9b	xmm9	ymm9
r10	r10d	r10w	r10b	xmm10	ymm10
r11	r11d	r11w	r11b	xmm11	ymm11
r12	r12d	r12w	r12b	xmm12	ymm12
r13	r13d	r13w	r13b	xmm13	ymm13
r14	r14d	r14w	r14b	xmm14	ymm14
r15	r15d	r15w	r15b	xmm15	ymm15

Appendix B

Side-Channel Patching

B.1 Example Patching Loop with GPT-4

```
1 System prompt:  
2 You are an expert at implementing constant-time cryptographic algorithms in C. Patch the  
given functions according to user's instructions. Do not give detailed explanations. The  
generated code should be complete, do not omit any part of the code. It should be able to  
run without any post-processing. You can implement new functions and integrate them with  
the original function. Do not introduce new arguments to the given function. Do not  
change the name of the function.
```

```
1 User prompt:  
2 int memory_leakage_case_1(int x, int y, int option) {  
3     volatile int z[3] = { 0, 2, 300 };  
4     z[2] = y;  
5     if (option > 3) {  
6         return z[1];  
7     } else {  
8         return z[x % 3];  
9     }  
10 }  
11  
12 z array is accessed dependent on the secret in line return z[x % 3];. Patch the code  
such that the array access is made input independent.
```

```
1 Response:  
2 Sure, I will patch the code using a technique called "array blinding", which uniformly  
accesses every element in the array to ensure that access times are input-independent.  
3  
4 Here is the patched function:  
5  
6 """C  
7 int memory_leakage_case_1(int x, int y, int option) {  
8     volatile int z[3] = { 0, 2, 300 };  
9     z[2] = y;  
10    volatile int result = 0;  
11    int i = 0;  
12    for(i=0;i<3;i++){
```

```

13     int equal = (i == (x%3));
14     result = (equal * z[i]) + ((1 - equal) * result); // if i == (x % 3), result will
15     be z[i], otherwise, it remains the same
16 }
17 if (option > 3) {
18     return z[1];
19 } else {
20     return result;
21 }
22 """
23 In the updated code, regardless of the input, the function now accesses all elements in
the array 'z'. We add a mask 'equal' which is 1 if the current index 'i' is the required
index and 0 otherwise. So, the 'result' will take the value of 'z[i]' only when 'i'
equals '(x % 3)'. This ensures all elements are accessed, preventing timing attacks based
on conditional access to array elements.

```

1 User prompt:
2 The condition in if (option > 3) is secret dependent and causes side channel
vulnerability. Patch the code such that it does not require any conditional execution.

1 Response:
2 Absolutely, I understand. We can eliminate the conditional by using a masking technique
similar to what we applied for the array access. Here is the patched function:
3
4 """
5 int memory_leakage_case_1(int x, int y, int option) {
6 volatile int z[3] = { 0, 2, 300 };
7 z[2] = y;
8 volatile int result = 0;
9 int i = 0;
10 for(i=0;i<3;i++) {
11 int equal = (i == (x%3));
12 result = (equal * z[i]) + ((1 - equal) * result);
13 }
14
15 int mask = (option > 3);
16 return (mask * z[1]) + ((1 - mask) * result);
17 }
18 """
19 With this new method, the function performs the calculation 'z[1]' and 'result' in every
case. The final returned value depends on the mask value. If 'option > 3', mask will be
'1' and 'z[1]' is returned. Otherwise, 'result' is returned. The code is now constant
time and no branches are secret dependent.

B.2 Microbenchmark of Leaky Functions Compiled from the Literature

```
1 // taken from Pitchfork, Cauligi, et al.
2 int memory_leakage_case_1(int x, int y, int option) {
3     volatile int z[3] = { 0, 2, 300 };
4     z[2] = y;
5     if (option > 3) {
6         return z[1];
7     } else {
8         return z[x % 3];
9     }
10 }
11 // table lookup - from DATA - Weiser et al.
12 unsigned char LUT[16]={0x52, 0x19, 0x3E, 0x7F,
13                         0x0C, 0x5A, 0x6D, 0x2B,
14                         0x3F, 0x1A, 0x7E, 0x53,
15                         0x6C, 0x5B, 0x0D, 0x37};
16 int memory_leakage_case_2_transform(int kval) { return LUT[kval % 16]; }
17 int memory_leakage_case_2(int key){
18     int val = memory_leakage_case_2_transform(0);
19     val+=memory_leakage_case_2_transform(key);
20     return val;
21 }
22
23 // from CacheD paper- Wang et al
24 int memory_leakage_case_3(int secret){
25     int table[128] = {0};
26     for (int i=0; i<128; i++){
27         table[i] = i;
28     }
29     int i, t;
30     int index = 0;
31     for (i=0; i<200; i++){
32         index = (index+secret) % 128;
33         t = table[index];
34         t = table[(index) % 79];
35     }
36     return t;
37 }
38 const uint8_t book[10] __attribute__((aligned(64))) = { 52, 48, 55, 51, 56, 54, 50,
39                                         49, 57, 53 };
40 uint8_t* memory_leakage_case_4(uint8_t* msg, unsigned len) {
41     for (unsigned i = 0; i < len; ++i)
42         msg[i] = book[msg[i]-48];
43     return msg;
44 }
```

```

1 // getelement-taken from CacheAudit, Doychev et al
2 unsigned int A[16] = {0, 1, 2, 3, 4, 5, 6, 7,
3                      8, 9, 10, 11, 12, 13, 14, 15};
4 int memory_leakage_case_5(int secret) {
5     if (secret < 16)
6         return A[secret];
7 }
8
9 // isDiffVul1 - taken from FlowTracker https://dl.acm.org/doi/pdf
10 // 10.1145/2892208.2892230
11 int branch_leakage_case_1(char *pw, char *in) {
12     int i;
13     for (i=0; i<16; i++) {
14         if (pw[i] != in[i])
15             return 0;
16     }
17     return 1;
18 }
19
20 // InsertionSort-taken from CacheAudit, Doychev, et al.
21 uint8_t * branch_leakage_case_2(uint8_t *a, int array_size){
22     int i, j, index;
23     for (i = 1; i < array_size; ++i){
24         index = a[i];
25         for (j = i; j > 0 && a[j-1] > index; j--)
26             a[j] = a[j-1];
27         a[j] = index;
28     }
29     return a;
30 }
31
32 // eq - Time variant - taken from FlowTracker https://dl.acm.org/doi/pdf
33 // 10.1145/2892208.2892230
34 int branch_leakage_case_3(char *p, char *q) {
35     if (p[0] != q[0])
36         return false;
37     else if (p[1] != q[1])
38         return false;
39     else
40         return p[2] == q[2];
41 }
42 // example 1-from Blazer, Antonopoulos, et al.
43 int branch_leakage_case_4(int high, uint low) {
44     int i;
45     if (high == 0) {
46         i = 0;
47         while(i < low) i++;
48     }
49     else {
50         i = low;
51         while(i > 0) i--;
52     }
53     return i;
54 }
```

```

1 // example 2-from Blazer, Antonopoulos, et al.
2 int branch_leakage_case_5(int high, int low) {
3     int i;
4     if (low > 0) { // O(2*low)
5         i = 0;
6         while(i<low) i++;
7         while(i>0) i--;
8     } else { // O(1)
9         if (high == 0) { i = 5; }
10        else { i = 0; i++; }
11    }
12    return i;
13 }
14
15 // taken from https://github.com/PLSysSec/haybale-pitchfork
16 int branch_leakage_case_6(int x) {
17     if (x > 10) {
18         return x % 200 * 3;
19     } else {
20         return x + 10;
21     }
22 }
23 // taken from https://github.com/PLSysSec/haybale-pitchfork
24 int branch_leakage_case_7(int x, int y, int option) {
25     volatile int z[3] = { 0, 2, 300 };
26     z[2] = y;
27     if (option > 3) {
28         return z[1];
29     } else {
30         return z[2];
31     }
32 }
33 // from ctgrind tool  github repo
34 char branch_leakage_case_8(unsigned char *a, unsigned char *b) {
35     unsigned i;
36     for (i = 0; i < 16; i++) {
37         if (a[i] != b[i])
38             return 0;
39     }
40     return 1;
41 }
```

```

1 // mu - taken from SC-Eliminator https://dl.acm.org/doi/pdf/10.1145/3213846.3213851
2 // the C code of a textbook implementation of a 3-way cipher.
3 int32_t * branch_leakage_case_9(int32_t *a) {
4     int i;
5     int32_t b[3];
6     b[0] = b[1] = b[2] = 0;
7     for (i=0; i<32; i++) {
8         b[0] <= 1;
9         b[1] <= 1;
10        b[2] <= 1;
11        if(a[0]&1)
12            b[2] |= 1;
13        if(a[1]&1)
14            b[1] |= 1;
15        if(a[2]&1)
16            b[0] |= 1;
17        a[0] >>= 1;
18        a[1] >>= 1;
19        a[2] >>= 1;
20    }
21    a[0] = b[0];
22    a[1] = b[1];
23    a[2] = b[2];
24
25 return a;
26 }
27
28 // taken from https://github.com/PLSysSec/haybale-pitchfork
29 uint8_t branch_leakage_case_10(uint8_t* public_arr, uint8_t public_arr_len, uint8_t*
30     secret_arr, uint8_t i) {
31     uint8_t x = public_arr[i];
32     for (int j = 0; j < public_arr_len; j++) {
33         secret_arr[j] += x;
34     }
35     if (x > 10) {
36         return public_arr[0] + secret_arr[0];
37     } else {
38         return public_arr[1] + secret_arr[1];
39     }
}

```

```

1 // bubblesort- taken from CacheAudit https://www.usenix.org/system/files/conference/
2 // usenixsecurity13/sec13-paper_doychev.pdf
3 uint8_t * branch_leakage_case_11(uint8_t *a, int n){
4     int i, j, temp;
5     for (i = 0; i < n - 1; ++i)
6         for (j = 0; j < n - 1 - i; ++j)
7             if (a[j] > a[j+1]){
8                 temp = a[j+1];
9                 a[j+1] = a[j];
10                a[j] = temp;
11            }
12        return a;
13    }
14 // SelectionSort - taken from CacheAudit https://www.usenix.org/system/files/
15 // conference/usenixsecurity13/sec13-paper_doychev.pdf
16 uint8_t * branch_leakage_case_12(uint8_t *a, int array_size){
17     int i;
18     for (i = 0; i < array_size - 1; ++i){
19         int j, min, temp;
20         min = i;
21         for (j = i+1; j < array_size; ++j){
22             if (a[j] < a[min])
23                 min = j;
24             temp = a[i];
25             a[i] = a[min];
26             a[min] = temp;
27         }
28         return a;
29     }

```

Appendix C

RL-based μ Arch Vulnerability Exploration

C.1 Instruction Sets

Table C.1: Number of instructions per set used in the action space

Instruction Set	Count	Instruction Set	Count
ADOX_ADCX	8	AES	12
AVX	695	AVX2	286
AVX2GATHER	16	AVX512F_512	2192
AVX512F_128	1816	AVX512F_256	1940
AVX512F_SCALAR	584	AVX512DQ_128	247
AVX512DQ_256	281	AVX512DQ_512	357
AVX512BW_128	467	AVX512BW_256	467
AVX512BW_512	467	AVX512F_128N	23
AVX512DQ_SCALAR	44	AVX512CD_512	38
AVX512CD_128	38	AVX512CD_256	38
AVX512BW_128N	8	AVX512DQ_128N	8
AVX512DQ_KOP	18	AVX512BW_KOP	34
AVX512F_KOP	15	AVXAES	12
I86	809	I386	196
I486REAL	37	CMOV	96
PENTIUMREAL	5	I186	124
LONGMODE	24	LAHF	2
I286PROTECTED	26	I286REAL	10
FAT_NOP	3	RDPMC	1
PPRO	2	BMI1	26
BMI2	32	CET	2
F16C	8	FMA	192
INVPcid	1	CMPXCHG16B	2
LZCNT	6	PENTIUMMMX	129
SSE	97	MOVBE	6
PCLMULQDQ	2	RDRAND	3
RDSEED	3	RDTSCP	1
RDWRFSGS	8	FXSAVE	2
FXSAVE64	2	SSEMCSR	2
SSE2	264	SSE2MMX	6
SSE3	20	SSE3X87	2
SSE4	96	SSE42	25
POPCNT	6	SSSE3MMX	32
SSSE3	32	X87	119
FCMOV	8	FCOMI	4
XSAVE	6	XSAVEC	2
XSAVEOPT	2	XSAVES	4